ENIGMA SIMULATOR

By Jonathan Wong

Submitted in partial fulfillment of the requirements
For the degree of Master of Science in
Object Orientated Software Systems

Approved by: ______________________
Dr. Peter Smith

Department of Computing
City University
London EC1V OHB

Abstract

One of the main problems in attempting to describe a WWII Enigma is their lack of availability. Since the end of War, nearly all of the many Enigmas in use have either been, lost, destroyed or fallen into a state of disrepair.

This project investigates the workings of a typical 1938 Naval Enigma and uses the Java language to recreate a web deployable applet that simulates those actions allowing users to encipher and decipher messages as the German forces did during the Second World War.

Table of Contents

1. Introduction to the Enigma Simulator

The Enigma machine was a device widely used by the Germans during the Second World War to encrypt radio messages sent to one another. The machine has gained public notoriety since WWII with the publication of numerous pieces of fictional and non-fictional literature as well as some less than impressive Hollywood productions. But when asked on the principals of its workings, very few people knew much about the machine. Most people knew it was a machine and some more knew it looked like an old fashioned typewriter, but that it seems is all that is known. This may be due to how any work involved with Enigma was always shrouded in secrecy. After the war, the thousands of people working on cracking Enigma at Bletchley, still bounded by the official secrets act simply disbanded. As time has passed and lessened the relevant importance of the Enigma to today's intelligence, official documents have been released allowing renewed interest possible, sadly many of the Enigma machines have long since disappeared hindering any curiosity that might exist.

Yet when looked into further, the Enigma was no more than a primitive electro-mechanical device used to perform simple poly-alphabetical shifts, (as opposed to non poly-alphabetical shifts where a letter is only mapped to one other letter).

For this reason the following project attempts to recreate an Enigma simulator that will both accurately represent the workings of the machine and give the user a clearer understanding of what factors contributed to the encoding of an enigma message. Hopefully, I will be able to break down the encrypting components into simplified units and show how unspectacular they are individually but how collectively they combine to create billions upon billions of combinations.

1.1. Deviations in the Enigma Project

The purpose of this project was not simply to build an Enigma Simulator but to expose its weaknesses. There were a variety of ways to exploit an Enigma, and two methods that were deemed historically successful were the Bombe and the Banburismus (see chapter2: Literature Review). Ideally it would have been a more exciting a prospect to build the Bombe, as there already exists Enigma simulators, which I shall talk about later.

The Bombe was a counter electro-mechanical device modelled on Enigmas that worked backwards from a set of enciphered text and its assumed plain text (the crib), ruling out settings from which the plain text could be enciphered to the enciphered text.

But as with any semi-complex problem, it is impractical to devise a solution without first fully understanding the problem. Also, the Bombe used a set of wheel components based on the Enigma to test for settings, in which case it would have been necessary to build large parts of an Enigma anyway. In short, I'm not sure I could have built a Bombe without building an Enigma first.

Unfortunately, with the time frame I was given and the constraints and commitments faced personally, the Bombe and Banburismus are only included as part of my research into the Enigma and its solutions. Instead more time was devoted to the Enigma to remove all bugs and provide greater usability.

It was only after a few weeks into designing an Enigma simulator that I discovered several simulators already existed, two I shall refer to belong to Andy Carlson(Fig.1) and Russell Schwarzger(Fig.2). Testing both Enigmas, I was quickly aware that the two simulators gave different answers despite looking similar in design. It was at this point that I felt that my design plan needed to be re-evaluated and altered to prevent myself ‘reinventing the wheel’. After some time spent investigating the two Enigma simulators and trying to figure out why they gave different outputs I became aware of the lack of information conveyed by the two machines. The two machines looked the same because they looked like an Enigma, but their respective inner workings must have been different for the results to be so (with all other things being equal, i.e. the Enigma settings). This was the only deduction I could make from each of the two machines. However, if I could visually demonstrate the paths the electrical signal took passing through the machine I could at least justify any answer my simulator gave rather than simply churning out possibly meaningless (and even random) characters. This would serve the purpose of improving understanding of the machine and increase the scope beyond the two existing machines.

To show the paths of the signal passing through an Enigma would have conflicted with an original requirement o that it should look authentic, as the workings in an Enigma are all hidden, but given the evidence of already existing “authentic looking” Enigmas I felt it appropriate and justified to modify my designs.

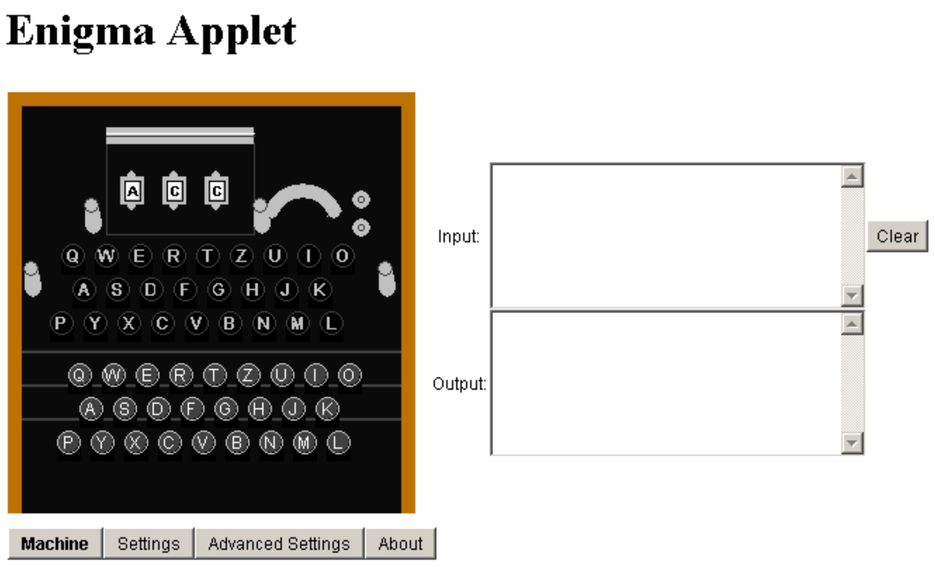
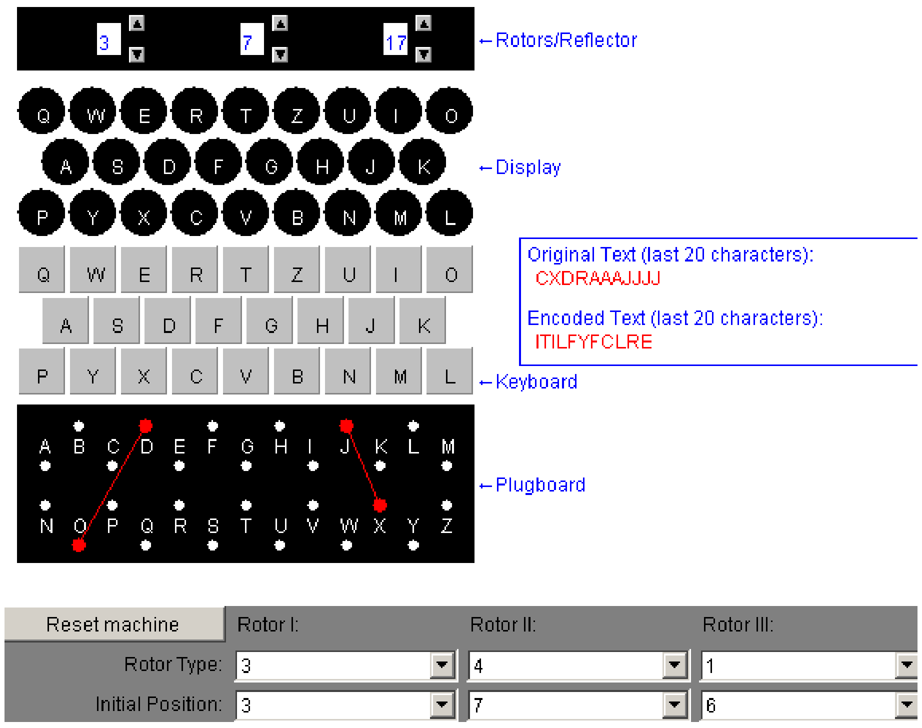


Figure 1. An Enigma Simulator written by Andy Carlson



Original Text (last 20 characters):
CXDRAAAJJJJ

Encoded Text (last 20 characters):
ITILFYFCLRE

	Rotor I:	Rotor II:	Rotor III:
Rotor Type:	3	4	1
Initial Position:	3	7	6

Figure 2. Another Enigma Simulator written by Russell Schwarzger

As previously mentioned much of the hoped building of the Bombe was left untouched and probably best left as a separate project for another student to undertake, instead I invested more time in learning the various swing packages to enrich the look and feel of my simulator. My reasoning behind this was to produce a high standard of graphical user interface that improved on my earlier prototypes, whose opinions ranged from visually poor and inadequate information. To achieve this required me make use of error prevention, the extension, overloading and overriding of components from both the AWT and Swing packages and the mapping of event listeners that would allow instantaneous updating of component's statuses with one another.

This addition to the project in turn conflicted with an earlier reasoning that the program be written as an applet. I found during mid-development after I introduced more elements of Swing that the applet became less accessible to an increasing number of browsers. Accessibility being the underlying factor for deployment as an applet I had to re-evaluate and resolve the conflict of these requirements. Further investigation in the Java newsgroups showed that the only way I would have been able to cater for all browsers was to use only JDK1.1 and incorporate no elements of Swing. I therefore decided that in order to make the simulator more attractive and user-friendly I would have to make the installation of a Sun Java Run-Time Environment a requirement to view the applet.

Going back to the question I asked about which of the two previous Enigmas were accurate, it occurred that this question would be asked of mine. The definitive answer of course would be to compare messages encrypted by those simulators with an actual Enigma machine. For lack of available real life Enigmas, this is just not feasible.

But, given that the action motion of an Enigma was relatively simple and consistently reported, the most likely source of inaccuracy was in the mappings of each wheel, for authenticity I wanted an accurate source for which to base my wheel mappings on, but in theoretically, so long as there existed a mapping for each of my wheel it should not affect the enciphering ability of my simulator.

1. To prove that any machine is consistent with itself a message must be able to encrypt and decrypt itself using the same base settings. This would be the most fundamental requirement of any Enigma machine and will be the first test I shall perform.
2. As previously mentioned, the only reason why two simulators should produce different results would most probably lie in the order of the mappings for each wheel. If I used the same mappings as any other Enigma, my results should be identical. For this reason I have chosen to use the same wheel mappings as Andy Carlson's Enigma so I could compare my results with an existing machine.

The question that will be looked at in greater detail asks how a mechanism can be devised to crack a message encrypted using an Enigma? But this first requires the question as to how an encoder encrypt messages using Enigma, which takes us back to the question: how does an Enigma perform encryption?
I hope to be able to answer these questions and hopefully demonstrate them clearly by the end of the project build.

To recap the points of my proposal:
Project Objectives:

Build a fully working enigma machine that allows the user to fully understand the workings of an enigma machine

Give the user a clearer understanding of the actions of an Enigma machine than those offered by existing simulators.

Answer Questions

how did the enigma generate so many combinations?
how were solutions to the Enigma found?

Areas to research

Anatomy of the wheels, the motion and the stecker
The operator procedure
Java Foundation Classes

The remainder of the project report chapters can be summarised into the following,

2. Looks at the history, evolution and anatomy of an Enigma machine, using descriptions given in books written on the subject and the recent set of documents released by the PRO.
3. Method – The process of my analysis and design, based on the requirements set out for the Enigma.
4. Program(documentation)Introduction to the system, an explanation of its functions and features including a quick tutorial.
5. Results - How does my simulator compare with the results of other simulators and whether there are any inconsistencies with the machine itself? How well does it explain the operations of the Enigma machine? Is the machine user-friendly?
6. Conclusion - What can be deduced from the project, what improvements could be made and where the project could go from here.

2. Introduction to the Enigma

Several notable Mathematicians worked on solutions to Enigma problem during its famous period and since then a number of authors have been so intrigued in the subject, that they too have devoted large parts of their lives to study the Enigma.
As part of my Literary Review I shall be discussing some of their publicated material relating to the Enigma's early conception, anatomy, mechanism, contentions in the mechanism, how they were used and how different versions of Enigma were solved.

2.1. Pre-War History of the Enigma

In brief, the Enigma was a machine that performed polyalphabetical transformations on letters, using a set of rotors (or wheels) which shifted after a set number of key presses, allowing electrical signals to light up letters indicating their cipher. In Slawo Wesolkowski's paper titled, "The Invention of Enigma and How the Polish Broke it Before the Start of WWII", he sets out various milestones in the Enigma's history before the advent of war and how several inventors contributed to the construction of the machine before Arthur Schreibus (who is identified in history as the man who invented the machine). It seems Schreibus' main contribution was the adding of a rotor principle to a electrical coding machine invented by Koch, but the idea of switching electrical signals between key presses had already been invented and patented by an American named Heburn, who ironically or perhaps pioneeringly sold his machine to the US Navy in 1928. Wesoklowski highlights what this type of machine achieves, sustaining the enemy's inability to decipher a message even if the encoding machine is captured.

After this early anthology of the Enigma, Wesoklowski writes of how the Poles played such an important part in breaking the Enigma, and how they even came into possession of an Enigma replica. In particular he talks about the set of Polish mathematicians from Poznan University, who through their talent for cryptography were recruited to join the Cipher Bureau, where given a replica of an Enigma, they were able to devise ingenious solutions and machinery that could break the early Enigma ciphers quickly.

One of these prominent Polish mathematicians was Marian Rejewski, who himself was in mid writing of a book on how he and his fellow Poles first cracked the Enigma ciphers back in the early 1930s. Sadly, he died before he could complete work and the book has never been published. Fortunately however, he did manage to publish short parts of his work throughout his life, some of which I will later refer to.

2.2. Anatomy of an Enigma

The latest batch of documents written by Turing (PRO 2000 HW 25/3) and released by the public records office serve as a good introduction in which to describe the Enigma machine.

In them he outlines the basic anatomy of an Enigma machine with a series of hand drawn sketches and notes describing the machine in its simplest, original form; 'unsteckered', referring to the absence of the later added steckerboard, which performed the function of swapping letters from the keyboard with one another, before and after the electrical signal went through the scrambling wheels.

The notes are extremely concise and one can only assume their purpose were to transmit understanding of his work to his superiors or his own staff. In them Turing gives a step-by-step run through of an unsteckered Enigma machine by describing it in 3 stages.

2.3. The electric circuit of the machine without the wheels

Firstly, without considering any of the scrambling wheels, there is a fixed entry disk (Eintrittwalze-EW) on the right hand side machine, which like the other discs is cylindrical with 26 contacts. The purpose of such a disc is to pair electrical signals (and therefore letters), depending upon the order and setting of the wheels.
In his description of the wheel he writes: "Notice that if F is the result of enciphering G, then G is the result of enciphering F at the same place, also that the result of enciphering G can never be G." Though he doesn't go on to say it, this epitomises the effect the reflector plate has on the Enigma machine. It keeps an electrical signal sent symmetrical but does not allow a signal to reach and depart the reflector plate using the same signal path.

2.4. The circuit through the wheels

Turing goes on to describe the remaining wheels, one of which (the reflector plate,) on the left hand side doesn't rotate but rigidly bounces the signal back as constant pairs. This gave the machine a symmetrical effect and the ability to code and decode the same message on any Enigma at the same start settings.
The three other wheels are each rotatable and removable, allowing interchangeability, so that guessing the correct order of the three wheels is a one in six chance. Later when extra two wheels were added, guessing the correct order meant picking the correct order of 3 from 6, this reduced the probability to one in sixty.
A clearer diagram of a wheel than that drawn by Turing, and taken from the web site of the former curator of Bletchley Park (Tony Sale), illustrates the make up of an Enigma wheel. On the right of each wheel there are 26 plate contacts and 26 spring contacts on the left. The spring contacts on the left hand side are there to make contact with the plate contacts of the next wheel on the right.
Each wheel has an inner wiring, which determines the mapping of letters, this is possibly the best way to think about the Enigma wheel, for "It is the core which effects the essential alphabetic substitution" (Tony Sale).
As an accompaniment below, I've included a diagram taken from one of Rejewski's papers written on the Enigma (Rejewski 1980), showing the reflector plate on the left, the entry wheel on the right and the moveable wheels and their mappings.

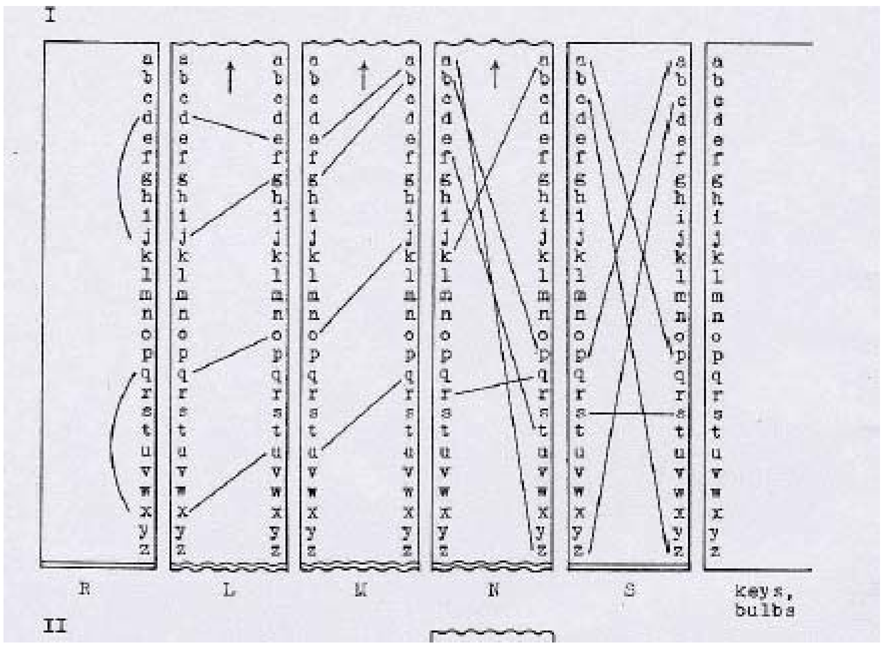


Figure 3. Representation of the wheel and reflector mappings as seen by Rejewski

2.5. The mechanism for turning the wheels and describing their positions

Turing completes his description of the Enigma machine by explaining the Enigma's window position - the set of three letters visible on top of the Enigma machine. This is not to be confused with the ring position (whose description is better served by Tony Sale see later).
Whilst describing the actions of a key press, he writes: "When a key is depressed the window position changes, but does not change further when the key is allowed to rise" (PRO 2000 HW 25/3), though Turing doesn't say explicitly in this document, it seems that the rotation of the wheel happens before the key press, this does tally up with the observations made by Shayler who explains that the rotation of the wheel happens prior to the electrical signal being sent. So pressing a letter with wheel positions AAA would send a signal through positions AAB.

The set of notes written by Turing appear to be a rough draft to the first chapter of his 'Treatise on Enigma' (Turing 99), and consists of a set of retyped documents that account Turing's own work during his time at Bletchley. These too have only been recently released into the public domain by the US National Archives.

The brief notes released by the PRO, offer a fascinating insight into the work that Turing did, but when it came to researching aspects beyond the Enigma's basic wheel, the explanation of the 'ring setting' or ringstellung was insufficient in that particular document. It is important to understand that the ring setting is not the same as the (initial) window setting, this distinction can be blurred and create problems when trying to visualise the wheel(s) in any subsequent solutions. Tony Sale gives a brief and comprehensible definition to the ringstellung in his web site as follows: "Position of alphabet-bearing tyre on wheel of Enigma machine, defined by number or letter at which a clip is set". From this, I found it easier to understand how inside each wheel there is a rubber ring whose settings can be turned in respect to the core of the wheel.
Sale adds that the Ring setting does not perform any additional scrambling, since any transformation it performs is in relation to all the letters, but what it does achieve is

change the turnover point for the carry mechanism. This point is repeated later when trying to deal with the ring settings whilst considering how to break an Enigma code setting.

2.6. The double Stepping of the middle rotor

Andrew Hodges (see later) and many other authors give the probability space of an Enigma's wheel setting as 26x26x26, but what they all fail to take into account is the phenomenon of the double stepping of the middle rotor as written about by David H. Hamer[97].

In this short paper he identifies not only how the phenomenon exists but also why it does so, the reason being inherent to its pawl-ratchet mechanism. The paper explains how each rotor has 26 ratchets on its right and a spring-loaded pawl on the left. What determines whether a rotor rotates is whether a stepping pawl can engage its ratchets. Which generally (except for the case of the double step) is where each rotor has its 'notch'. Where this predicted behaviour becomes wry is when the second (middle) rotor has completed its rotation and its ratchets are still being engaged when they should be released. Instead of remaining on the new setting for another complete rotation of the wheel to its right (the fast wheel), the middle wheel would only remain on that setting for one further key press before moving onto the next setting which it would remain on for a complete revolution.

In the reproduced example the middle rotor remains on setting E for only the next key press instead of the predicted next twenty-six. According to Hamer (*ibid.*) the probability space is therefore 26x25x26.

(Observer rotor movement)			(Actions causing the rotor movement)		
1	2	3	1	2	3
A	D	O			p3/r3
A	D	P			p3/r3
A	D	Q			p3/r3
A	**E**	R		p2/r2	p3/r3 and p2/n3
B	F	S	p1/r1	p1/n2	p3/r3
B	F	T			p3/r3
B	F	U			p3/r3

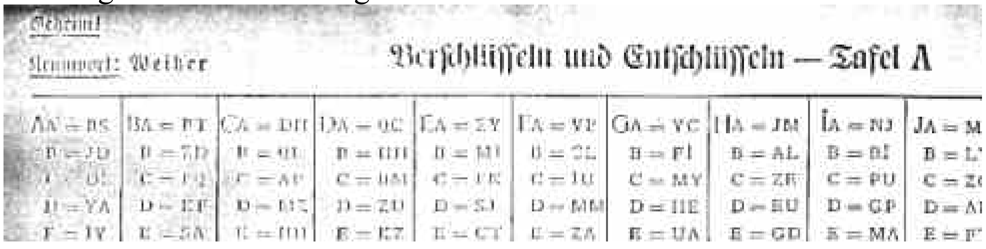
2.7. How Enigmas were used

As with his history of the Enigma machine, Hugh Sebag-Montefiore gives a similarly detailed narrative of the Enigma's timeline from the perspective of the operator, chronicling the changes that he/she needed to implement while operating the machine throughout the period.

How the Enigma was used varied not only through time, but also through the divisions of the German armed forces. As a pre 1937 naval example, on any given day, an Axis operator would be given the stecker settings, wheel order and the ringstellung with which he must use, but crucially not the starting (window) positions.
For this the sending operator would have to pick six letters, three for the starting positions and three for the settings in which to code/decode those starting positions.
It was this freedom to allow the operator to choose from a wide as possible variable space and then the operators subsequent lack of imagination or plain idleness to then go on and

pick a diverse set of letters (such as the 1st six letters on the top row of the keyboard) that allowed an entry point into the Enigma for the code breakers.

However, after 1937, the Naval Enigmas ceased to use this system and instead required the encoding operator to select two sets of three letters (trigram) from a book (sometimes referred to as the Kenngruppe or K-book), before adding two bogey letters, one before the first triplet and the other after the second triplet, the now two sets of four letters were then manipulated into a formulated set of pairs which were then substituted using a monthly book of bigram tables, before being sent out with a set daily indicator to make up the key for that day, thus preventing the unimaginative or lazy enciphering of keys and ensuring that the Naval Enigma remain the hardest to break.


Geheim!

Kennwort: Wetter

Verſchlüſſeln und Entſchlüſſeln — Tafel A

AA = BS	BA = FT	CA = DH	DA = QC	EA = ZY	FA = VP	GA = YC	HA = JM	IA = NJ	JA = M
B = JD	B = ZD	B = [illegible]	B = [illegible]	B = MI	B = CL	B = FI	B = AL	B = BI	B = L
C = [illegible]	C = [illegible]	C = AP	C = HM	C = [illegible]	C = IU	C = MY	C = ZR	C = PU	C = Z
D = YA	D = EF	D = [illegible]	D = ZU	D = SJ	D = MM	D = HE	D = EU	D = GP	D = A
E = IV	E = SA	E = [illegible]	E = KZ	E = CT	E = ZA	E = UA	E = GD	E = MA	E = F

Figure 4 German Bigram Tables (taken from Tony Sales' Bletchley Park web site)

2.8. How Enigmas are solved

Solutions, given by different texts and authors vary depending upon which period of the Enigma is being considered. Overall, Sebag-Montefiore gives a well-written account of all the main methods used and accompanies each with a fairly detailed appendix. It was hard however, to be able to visualise thoroughly any of the solutions using just this text alone (or any single source).

Ultimately, I've found that the best way to think of an Enigma's solution is of a point in an Enigma machine's setting which will produce the required transformations. The 'required transformations' refer to the mapping of a cipher text to its equivalent (or at least suspected equivalent) plain text. From any of Rejewksi's or Turing's descriptions on the Bomba or Bombe, it seems these required mappings or 'cribs' as they were called, formed a fundamental base upon which to perform tests which either refuted or confirmed whether a set of hypothesised Enigma settings would allow the mapping of the observed cipher text with the assumed plain text.

Using the factors stated in the US 6812 Bombe Report 1944 (Tony Sale 2000, 3), an Enigma message is solvable if the following information can be known about the message when it is first encoded:

1. The Stecker Settings (plug board)
2. The Scrambler
   2.1 The Wheel Order
   2.2 The Ring Settings
   2.3 The Initial Wheel Settings (or window settings or indicators)
3. (Naval enigma – additional keycode – used to scramble the window settings)

In this reformatted government document, amongst the markings of "top secret", there are ninety pages giving a manual style solution to this very Enigma problem, explaining the use of the Bombe and how bigram tables should be reconstructed. Admittedly, I found it difficult to understand these methods without first tackling an Enigma problem that assumed no steckering.

Andrew Hodges does this his book by writing, "Supposing that it is known for certain that UILKNTN is the encipherment of the word GENERAL by an Enigma without a plugboard. This means there exists a rotor position, such that U is transformed to G, and such that the next position transforms I into E, the next position, K to L etc. There is no obstacle in principle to making a search through all the rotor positions until this particular pattern is found" (Hodges, 179).
He goes on to explain that following on from this principle, you could set up seven machines each with incremental stepped consecutive positions from the initial position you are testing for i.e. if for wheel setting 1,2,3 you wanted to see whether the initial starting position AAA transformed GENERAL to UILKNTN then you could set up seven Enigmas with positions: AAA, AAB, AAC, AAD, AAE, AAF and AAG and test whether the 1st Enigma machine churned out letter U, the 2nd Enigma churned out the letter I, and so on. If this weren't so then the machines would be set to the next testing position AAB, (our 1st Enigma machine would be set to this and test whether G transforms to U, simultaneously the 2nd machine would be set to AAC to see whether E transforms to I and so on).

In fact, this process describes that of a Bomba (*ibid.* 175). As might be seen from the above example, this only looks for solutions for one particular wheel setting. Which before 1937 meant with only 3 possible wheels, 3x2x1 = 6 possibilities. Therefore if 6 bomby were built each consisting of its own set of Wheels, one could make all the tests simultaneously for all the wheel order settings, and it seems from all accounts that this is what the Polish crypto-analysts did.
However, the later introduction of an extra 2 wheels (before there were eventually 8 on the Naval Enigma) meant 5x4x3 = 60 possibilities meaning 60 of these bomby were required, shifting the problem of solving the Enigma into a logistical problem.

2.9. Other Methods

The main methods that are generally talked about when trying to solve an Enigma cipher will likely include the Characteristic, Bombe, Banburismus and Rodding methods. Given that there were 10,000 people worked at Bletchley alone (by the end the war), and assuming not all of them were cipher clerks, it is quite reasonable to assume that there were many more methods devised to break an Enigma cipher. As each method is in itself a fairly sizeable topic I will narrow my review to focus on giving a brief description of text relating to the Bombe and Banburismus.

I found the Bombe section of Graham Ellsbury's website (Ellsbury 1998) to be an invaluable source and introduction to this topic. It is well known that the Bombe was an electro-mechanical device that was similar to the Polish Bomba, but where the main differences in the two machines lay were in that the Bombes were able to look at all 60 possible rotor settings and had the added ability to refute or confirm hypothesised stecker settings. The machine is credited to Turing due to the principle he employed in the machine by assuming a set of steckered pairs of a crib and then through a process of proof by contradiction eliminating impossible set-ups.

Nevertheless the Bombe was still required to perform an exhaustive search of the possible combinations in which the machine could be set up

(107,458,687,327,300,000,000,000 according to Ellsbury, though he too doesn't take into account the middle rotor factor).

Turing devised a technique named the Banburismus that could limit the number of wheel settings needed to be tested on a Bombe, by picking out different messages that were suspected of being coded using similar coded indicators. Tony Sales' web site has a marvellous java-script simulation of this, but the user interface is almost impossible to use until you've read about and understood the Banburismus.

The principle of the Banburismus is that messages encoded using the same Enigma settings will produce a frequency set of letters similar to one another. With the aid of perforated sheets to represent matching numbers of ciphered letters, cryptoanalysts could perform tests for incremental wheel settings by sliding sheets against each other looking for the maximum number of letter count matches (represented by an alignment of holes in the perforated Banbury paper). From this the relative message settings of two enciphering machines could then be worked out, and then using the knowledge of the differing turning points of each Enigma wheel, previously possible wheel positions could be ruled out. This method was particularly successful against the German Naval Enigma's use of Bigram and Trigram table and could reduce a set of 336 wheel order positions(8x7x6) to as low as 3, substantially minimalising the sometimes premium time required on a Bombe.

2.10. Summary

There are very few differences in the way the Enigma is reported, but reading the vast amounts of historical and technical reports now available, it is evident that there were a multitude of ways in which an Enigma was used and even more ways to solve them. Some range from the relatively simple principle of exhaustive checking an Enigma until a desired outcome occurs while others required the more complex use of perforated paper and knowledge of Group Theory to eliminate many of the possible permutations. In fact there is no record of any single method in which to find a/the solution to an Enigma's setting without knowing them in the first place. The Banburismus, Rodding and Character and all other methods were ways in which to reduce the search space through knowing the oddities of the machine such as its inability to map a letter L to itself and the different turnover points of the earlier wheels. Inevitably, a 'solution' to the Enigma tended to be a process that reduced possible solutions. These possible solutions were a set of settings on the machine that didn't violate assumptions that were based on knowledge of the Enigma. Any possible solutions thrown up by the Banburismus for example, still needed to be tested on an Enigma machine, and it seems despite all the methods invented that help reduce the possible number of settings, this was the only way in which to verify an Enigma's settings belonged to those of an enciphered text.

3. Method

In this section I shall describe the process in which the Enigma simulator was built, following the thorough investigation into the Enigma's actions in the previous chapter. I shall examine the requirements, through to the workflow method and how individual components were eventually constructed and the reasoning behind the methods used.

3.1. Requirements

The main purpose of constructing a simulator is to create a system that acts and behaves in a way consistent to that which you are simulating. Bearing this in mind many of the requirements for the Enigma are straightforward and can be broken down into the following:

The simulator must be able to encrypt and decrypt messages using the same settings
The simulator shall be able to incorporate the behaviour of the window settings
The simulator shall be able to incorporate the behaviour of the inner ring settings
The simulator shall be able to incorporate the behaviour of the rotor orders
The simulator shall be able to incorporate the behaviour of the stecker settings
The simulator shall be able to demonstrate clearly the movement of the rotors
The simulator shall be able to demonstrate clearly the signal paths taken between a key pressed and the resulting illuminated light.
The will allow the user to change the wheel order
The will allow the user to change the window positions of each rotor
The will allow the user to change the inner ring settings for each rotor
The will allow the user to change the stecker settings of the Enigma simulator

Non-functional requirements
The simulator will be written as a Java applet and deployable on a web page
The system shall be aesthetically pleasing and professional in appearance.

3.2. Use Cases

The operator changes rotors
The operator changes the reflector
The operator changes the ring settings
The operator changes the window positions
The operator changes the stecker settings
The operator resets the output panel
The operator resets the input panel
The operator deletes his last key entry

3.3. Analysis – Finding classes

Trying to model a set of program components with corresponding machine components seemed a good way to begin. I started by describing an Enigma's movements through written sentences and using the technique of noun/verb analysis to extract a set of possible classes.

"*As the operator types a series of letters into the keyboard, a letter's signal first passes through the Steckerboard, if that letter has been wired to an alternate letter c' the signal will pass along the path of alternate letter c' instead of it's own. Once through the* *stecker, the right hand rotor shifts up one position prior to the signal passing it. If the right hand rotor shifts into its own 'carry position' the next rotor to its immediate left (the middle rotor) shifts also, if that in turn shifts the middle rotor onto its carry position then the last rotor on the left will also shift. This being the case all three wheels will shift collectively prior to the signal passing through it.*

A signal passing through a rotor will be mapped to a different letter depending on three factors. The wheel it's passing (there are six-eight), the shift position of the wheel at the time the letter passes through and the inner setting of the wheel.

Once through the three scrambling wheel the signal hits a fixed reflector wheel which has its own set of mappings and can be chosen from two wheels, before bouncing back along a different path through the three scrambling discs and stecker finally illuminating a light indicating the encoded letter."

The core elements that suggest being turned into classes are the rotors, keyboard, reflector and light board.

The rotor class after is probably the most complex and crucial in the role of the Enigma. It must keep track of its own window setting, ring setting and position within the Enigma machine. From the above CRC it seems evident that it must also be able to perform a shift operation to simulate the single rotations of a real machine.

Similarly the reflector is similar in principal to a rotor because of the mapping function it performs, however the difference is the reflector doesn't rotate and therefore doesn't have a carry position, nor does it have a ring position or window position. Because of these arguments it is debateable whether the rotor and reflector wheel classes can be combined or extended from each other.

The keyboard class will be a set of buttons used to input signals through the scrambler, while the light-board will highlight the end result depending on the outcome of the path it takes through the scrambler and steckers.

The above descriptions of the various class components at this point do not take into account their own display. To display the workings to a screen output as a graphical illustration, each of the above classes will also need to have some kind of paint method. The Rotor class would not only know its window positions, ring positions and any outcome of a letter being passed through its mappings, it would also have to graphically draw these to screen.

For this reason I took the decision to extend all of the above components from a JPanel class thereby allowing both the ability to draw on its respective inherited graphics context and to readily add them to the overall JApplet container class I would be using to place them in.

(Extending from JApplet would facilitate the loading of sound and image, and the abilty to place the simulator on a web page allowing greater access.)

3.4. Workflow Process

Because most of the classes like the components they represent can be modelled individually, the development process that I shall be following will be the iterative and incremental development cycle (spiral model). Apart from the list of classes and brief design of the applet I have already talked about so far, I find it extremely hard to conceptualise completely how the overall design will eventually look so early on.

The advantages of this method is that it allows me to focus on implementing a small but problematic subset of the system like the rotor, early on before assessing how to construct parts which later rely heavily on the premise that these work accurately.

For each sub-component such as the rotor, reflector, keyboard, my strategy would be to rapidly and iteratively construct prototypes, which would either be discarded or improved at the end of each sub-cycle.

Therefore, my plan to build the Enigma would be to build components individually around the loose model I have given above, incorporating parts together at the earliest possible point and my order of schedule is planned as follows:

I would first work on the Rotor class and instantiate three of them in series to partially represent the scrambler to test the path a signal took in one direction. I could compare this path with the description given by Carlson[1] before implementing a keyboard class so to allow the user to determine the character being entered into the scrambler. Once I had managed to perfect the movements of these components I would implement the Reflector class to send the signal back. Finally, I would add the remaining steckers, light panel, and information panels.

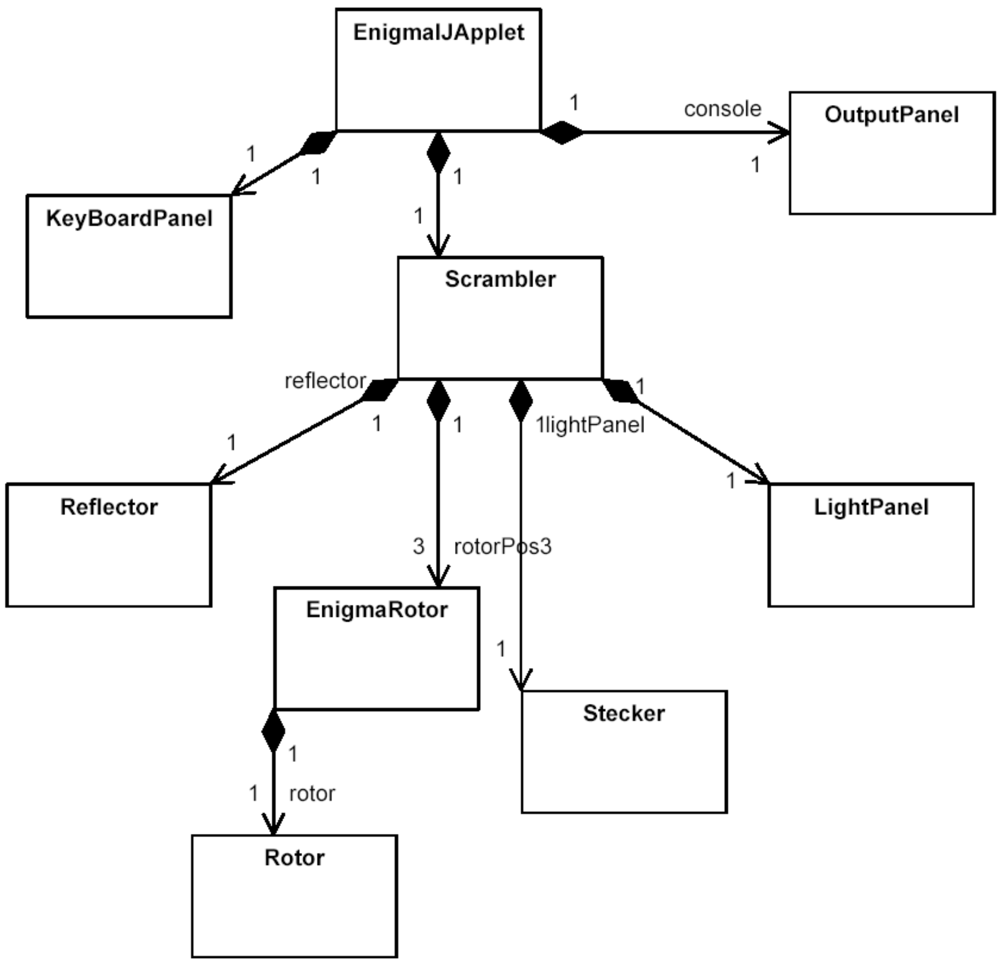


Figure 5. First Draft of the Enigma Simulator

This would be my basis to start with and as the build progressed I found it necessary to include additional classes.

3.5. Implementation

3.5.1. The Rotor class.

My aim was to reproduce the trace results talked about earlier (Carlson)
The results showed that entering the letters ABCDEFG(left column) into the Enigma with a wheel order of I,II,III, initial window settings AAA, inner ring settings AAA and no steckers would produce the output BJELRQZ(right column).

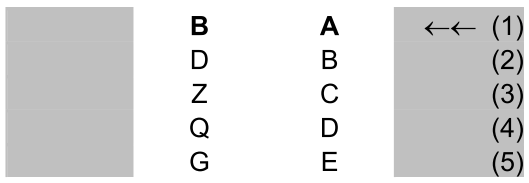
The description given by Carlson are reproduced below:

```
ABDC CCDD DDFF F->S SSSS SSEE EFCB
BDHF FFII IIVV V->W WWNN NNTT TVLJ
CFLI IIXX XXRR R->B BBWW WWMM MPHE
DHPL LLHH HHQQ Q->E EEAA AAAA AEPL
EJTO OOMM MMOO O->M MMCC CCPP PUWR
FLVP PPCC CCMM M->O OOMM MMOO OUWQ
GNNG GGRR RRUU U->C CCYY YYVV VCGZ
```

```
Rotor 1:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
            EKMFLGDQVZNTOWYHXUSPAIBRCJ

Rotor 2:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
            AJDKSIRUXBLHWTMCQGZNPYFVOE

Rotor 3:    ABCDEFGHIJKLMNOPQRSTUVWXYZ
            BDFHJLCPRTXVZNYEIWGAKMUSQO

Reflector:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
            YRUHQSLDPXNGOKMIEBFZCWVJAT
```

"The A is unchanged by the stecker. The keystroke advances the right rotor (3) BEFORE encryption, so the A is delivered to the B terminal on rotor 3. Rotor 3 changes it to a D. This is then passed across in the C position (after adjusting for the rotor offset from its neutral position) to the middle rotor (2). Both middle (2) and left (1) rotors are in their neutral positions so we can just apply the wiring transformations without worrying about offsets, so C becomes D and then D becomes F.
The reflector turns the F to S, then on the return path, the left rotor makes no change, the middle changes S to E (note that we are going in the opposite direction now). We then adjust for the offset, so the E activates the F terminal on the right rotor (3), which transforms it to a C. Readjusting for the offset gives terminal B. Passing again through the stecker gives no change." (Carlson)

Using these notes, a study had to be conducted on a wheel's movement. I did this by first considering a wheel without an inner ring (as this simplifies our model a little until later). This means that in the example below, whenever a signal reaches A from the right hand side (RHS), the signal will always leave the left hand side(LHS) via the letter B . And vice-versa, if the signal enters from the LHS via B, then the signal will leave through the RHS via point **A**.
Note, this is not the same as when a signal enters from the RHS via B, the signal that would then leave the LHS via **D**.



To recap: Entering the wheel below via B gives A if signal is from LHS
& D if signal is from RHS

(1)→→	B	**A**	(1)
(2)	D	B	←← (2)
(3)	Z	C	(3)
(4)	Q	D	(4)
(5)	G	E	(5)

But when used as part of an Enigma, the wheel shifts each time a signal is passed, thereby changing the contact point the wheel touches each entry contact point .

A	B	A	
B	D	B	←← (1)
C	Z	C	(2)
D	Q	D	(3)
E	G	E	(4)
			(5)

Figure 6. Window Position = B

As can be seen the output of pressing letter A(entry point 1) produces B in Figure 2, and D in Figure 2. Therefore any function that I write to determine the outcome of a letter passed through a wheel must take into account the window position.
If we take the default window position A as an integer value of 0, then it makes it simple to represent the change in window positions as a shift in the whole array.

At this point my function looked like this.

```
public char outputLetter(int n){
        if (shifUp)
                shiftUp()
                return (rotor[n + window_position]);
}
```

where rotor[] was my array of characters in the left column
{'B', 'D', 'Z', 'Q','G'}

After trying to pass the result of one wheel's output into the next, it became apparent that it would be more useful to pass an int as a parameter instead of a char. This was because when dealing with multiple wheels in series, an exit letter Z, which would be the input letter of the next wheel array wouldn't necessarily be the twenty-sixth letter of our array (as the above array shows). I later realised that the return type should have been an int as well for the same reasons.

Confusion began to arise before this because there was no clear distinction between letters and wheel positions. You could think of an element in an array mapping as either a numbered array element or as a character representing that array element.
Describing the process using only characters blurs the distinction between the entry point of the rotor element and the letter that resides at that element.
The letter E in the array ABC = {A,B,C,D,E,F……X,Y,Z} can be described as either the element ABC[4] or the element at entry point E. This letter is mapped to E (no mapping)

and this is the resulting letter. This works well for an ABC element but when considering the rotors whose characters aren't organised we develop confusion.
The letter E in the array rotor1 = {E,K,M,F,L……R,J,C} can be described as either the element rotor1[4] or the element at entry point E. Entry point E, is mapped to the letter F which means the exit point rotor1 and therefore the entry point of the next rotor rotor2, rotor[5] or the element at entry point F. As confusing at the above example is, without the references to exact numeric rotor elements the description becomes even harder to understand.

```
public int outputLetter(int n){
        if (shifUp)
                shiftUp()
                return (rotor[n + window_position]);
}
```

The problem with this method is that it doesn't actually work. The value being returned as an element of a rotor position, is a character, when the signature requires an int. What was needed was to convert that character into an integer ready to access the next rotor at the correct element (i.e. between 0 and 25).

Converting a letter to an integer ready to access the next element
Using the fact that the character represented by A = 65, B = 66 and so on up to Z =90.
I wrote a function to convert a letter into an integer to represent the exit/entry points in a rotor.

```
public int toNum(char letter){
        letter – 65;
}
```

3.5.2. Including the ring position

At this point I hadn't fully understood the ring position due to the lack of diagram and pictures showing the inner core of the rotor. The descriptions generally given were that it consisted of an inner ring, which could be rotated, in respect to the display of the wheel. As with most descriptions of the ring settings they first begin by describing window position, an example:

"*Obviously, the rotors had to be marked in some way on the outside so that the different positions (window positions) could be identified. However, here entered yet another element of complexity. Each rotor was encircled by a ring bearing the 26 letters, so that the with the ring fixed in position, each letter would label a rotor position. However, the position of the ring relative to the wirings, would be changed each day. The wirings might be thought of as labelled by numbers from 1 to 26, and the position of the ring by the letters A to Z appearing in the window. So a ring-setting would determine where the ring was to sit on the rotor, with perhaps the letter G on position 1, H on position 2, and so forth.*" (Hodges, 168)

Unfortunately, that was the best description I found for the inner ring-settings. I think the crucial part of the description was in the last line, and what I assume he's trying to say is that because the inner ring is rotateable against the outer wheel, once you change the selection (using an ordered alphabetical wheel ABC…XYZ) where 'A' is sitting on position 1 (and 'B' on position 2), to a new setting where 'A' sits on position 6, there is a relative shift of 6 positions for all of the letters. So even though the window position shows the letter 'A' you are in fact (six positions down) at position 'G'.

I did not actively test out the ring position, at this point so early on I hadn't worked out how if possible I could represent the inner ring setting, though I included it as I thought it should be. And, so long as I did not change its value from the default initialisation value 0 it would not affect the function's answer.

```
public int outputLetter(int n){
        if (shifUp)
                shiftUp()
                return (rotor[n + window_position – ring_position]);
}
```

3.5.3. **Drawing the result to screen**

Everyone who has ever written anything on the Enigma have himself or herself likely to have written an explanation in which the Enigma operates. It wasn't until I looked back over a paper by Rejewski (1980) that I was sure how I wanted to represent the rotors.

Rather than just assume wheel mappings as simple arrays to which an entry point is mapped, Rejewski's diagram(*ibid.*) clearly shows this by illustrating the entry point a letter signal makes with its corresponding contact point, the resulting letter mapped, the exit point and crucially the entry point of the next rotor. (Figure.1, page9)

The shifting pattern was also clearly demonstrated by the movement of the rotor strip as seen below. And because these strips were meant to represent a circular rotor it would be intuitive to wrap the overlap of any strip to the bottom or top.

My plan at this stage was to draw two rows of characters
Draw accurate lines mappings between the two rows of characters
Shift those letters to represent the changes in window positions
Preserve those line mappings even when the window positions changed

The following was later added to take into account the ring settings
Shift one side of the letters to represent the changes in the ring settings
Preserve those line mappings even when the ring settings change

Once I written the rotor class to perform the operations I wanted, I experienced difficulties in adding a set of buttons without ruining the display I had just created. My understanding of layout managers and techniques was less than perfect at this stage and looking back now I could have done this differently. Instead to get around the problem, I created another class called EnigmaRotor that also extended JPanel and with a BorderLayout in which to hold the rotor class I had just written along with the desired buttons.
A technique I began to use here and throughout many of my later classes was to override the getPreferredSize() method inherited from the JPanel.
This solved the layout problems I experienced with panels that would not size properly within containing panels and therefore display only partially. The setSize(int n, int m) method was used in vain.

3.5.4. Drawing the char array

Having already established a set of character arrays within the rotor class, I could make use of the array by calling on the drawChars method from the Graphics class to display to a JPanel object. By first creating a method called getNode(char c) and then passing elements of the mapping array (which is likely to be unordered) I found the class could keep track of the array position of each letter yet draw them in an ordered column.

```
for (int i = 0; i < rotor.length; i++){
        g.drawChars(abc, (ring + window+i)%abc.length,1,
        rightNode,yStep*((getInt(abc[(window + i)%abc.length])+26-window)%26)+yTopBuffer);

        g.drawChars(rotor, (window+i)%rotor.length,1, leftNode,
        yStep*((getInt(rotor[(window + i)%rotor.length])+26-window)%26) +yTopBuffer);
}
```

What the above function does, is to incrementally take the each element of an ordered ‘abc’ array, draws this on the right hand side and take its corresponding mapped element from the unordered ‘rotor’ array, draws that element from the rotor array on the left hand side (in its respective ordered position calculated using the getNode(char c) method) and draws a connection between those letters. The code for the connection is not shown above.

After several attempts I ended up with the following

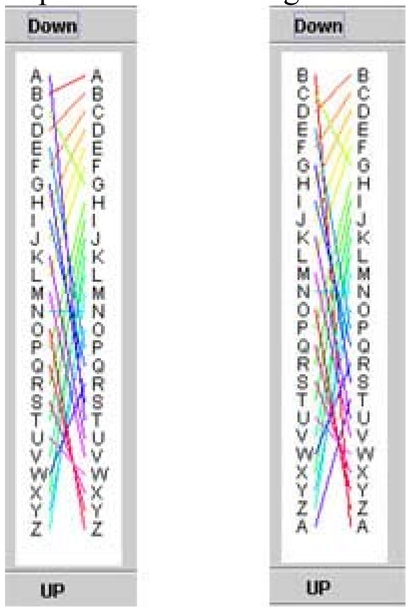

Down
ABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZ
UP
Down
BCDEFGHIJKLMNOPQRSTUVWXYZA
BCDEFGHIJKLMNOPQRSTUVWXYZA
UP

Figure 7. Mappings in rotor 3 which are preserved even when shifted

.

It is not at all clear here, but the top line of the rotor is meant to represent the window position of the rotor. Therefore the window position of the left rotor is ‘A’ and the result of pressing a key will shift the rotor by one so that the new window position is ‘B’(as shown by the right rotor). In the final version the window positions, ring positions and carry positions are highlighted by a box with the respective abbreviations: wp, rp and cp.

Once I was satisfied with this, I continued by adding three rotors in series within a new containing class again derived from JPanel and named it **Scrambler**. My objective was to have three distinct rotors working collectively using the result of feeding the 1st (right) rotor, as an input parameter for the next (middle) rotor, and similarly its result used as an input parameter for the (left) 3rd rotor.
I did this by grouping the methods together in the scrambler and using the result of one as the input for the next.

```
public void passLetter(int n){
        rotorPos3.getRotor().shiftUp();
                int t1 = rotorPos3.getRotor().passLetterL(sOut);
                int t2 = rotorPos2.getRotor().passLetterL(t1);
                int t3 = rotorPos1.getRotor().passLetterL(t2);
}
```

Using the passLetterL(int) method through each rotor not only returned a correct output value for that rotor, but also drew those changes. So that if the 1st trace t1 is passed through the 2nd rotor position (rotorPos2). The rotor at that position would calculate and draw the outcome of that letter being passed through it, taking into account its own window and ring position.

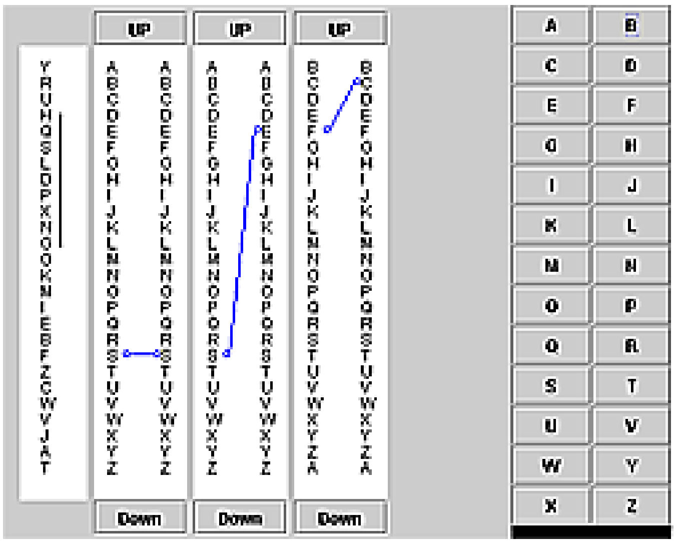


Figure 8. Three rotors in series

At this stage there was no automatic shifting of the rotors so the right rotor had to be shifted manually using its ‘up’ button. The letter B enters the right rotor at position 2 and makes contact with the Node C, this is mapped to letter F which is now at exit position 5.

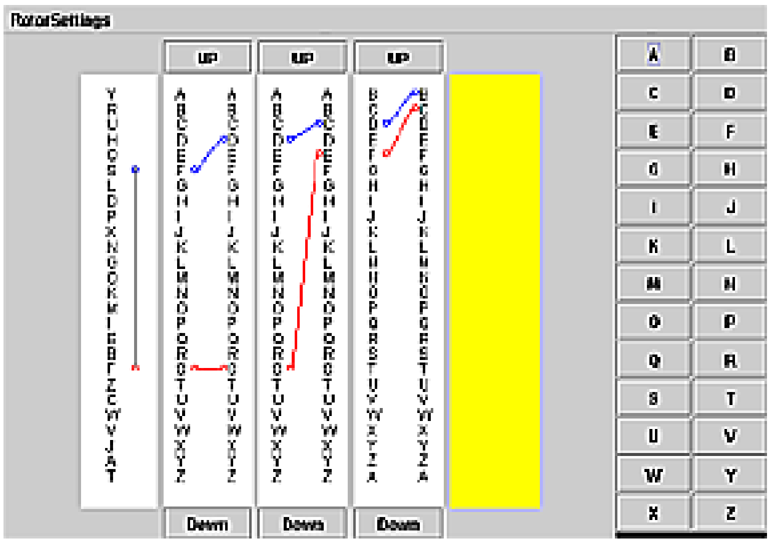


Figure 9. Demonstrates a path described by Carlson

Placing a method in the Scrambler class, which incorporated the systematic actions of all the rotors, solved the automatic shifting. Using a set of nested if statements, I could place a set of conditions within the method that would pass letters to determine which rotors if any should turn.

To recap, the right hand rotor (rotorPos3) rotates every time a letter is passed, but the middle rotor only turns when the right rotor reaches its carry position, whilst the left rotor only turns when the middle rotor turns onto its carry position. The following lines of code seemed consistent with my understanding of the rotor's carry position.

```
if (rotorPos3.getRotor().window == rotorPos3.getRotor().carryposition){
        rotorPos2.getRotor().shiftUp();

                if (rotorPos2.getRotor().window == rotorPos2.getRotor().carryposition){
                        rotorPos1.getRotor().shiftUp();
                }
}
```

The reflector was very similar to the rotor class in principal, its purpose was to perform character mapping and to reflect the signal back. The difference being that the reflector was fixed and did not rotate like the other rotors. Also, where rotors can be chosen from a set of three to eight, depending on which version of the Enigma machine being modelled, the reflector was always chosen from a set of two reflectors:

```
        char[] reflectorB = {'Y','R','U','H','Q',… ,'V','J','A','T'};
        char[] reflectorC = {'F','V','P','J','I',… ,'N','M','H','L'};
```

Looking back it would have been quite feasible to build an abstract class that represented the core mapping features of the rotors and the reflector, but it is arguable whether this would have resulted in a clearer design or any lesser amount of code.
Another difference became apparent after I realised that the signal needed to be reflected back through the same rotors I had already passed a signal through. Whereas the reflector only needed to perform a mapping in one direction, a rotor needed to calculate and draw a signal coming initially from right to left before returning back from left to right.

There are a number of ways a signal returning can be thought about, some can be confusing. But the best way as always was to continue thinking about the wirings as mappings. If we have the array ‘Mapping’

```
ROTOR FACE  = {‘A’,’B’,’C’,’D’,’E’,’F’…’W’,’X’,’Y’,’Z’}
MAPPING     = {‘Q’,’W’,’Z’,’R’,’X’,’Y’…’C’,’B’,’A’,’L’}
```

Entering the rotor face from the right hand side at letter point ‘A’ would result in the exit at ‘Q’. But if we now apply the inverse and go back (i.e. from the other direction), entry from the left hand side at letter point ‘A’ no longer results in ‘Q’ but in ‘Y’.

The array ‘Mapping’ cannot be simply used directly for calculating the output when passing a letter in the opposite direction. But using this array we can construct an inverse mapping that can be used.
As stated passing letter A from the left produces ‘Y’, and you can see from the same direction passing ‘B’ produces ‘X’ and ‘C’ produces ‘W’ and so on until we have constructed an array mapping that represents the inverse of the mapping we want.

```
MAPPING_INV = {‘Y’,’X’,’W’…’E’,’F’,’B’,’C’}
```

This was done for all the six Enigma rotor mappings I was trying to simulate. It was quite a tedious task and relatively time consuming. A function could have been written to calculate the inverse rather than rewrite all the mappings, but I wanted to keep things as simple as possible and felt this was a way of doing so.

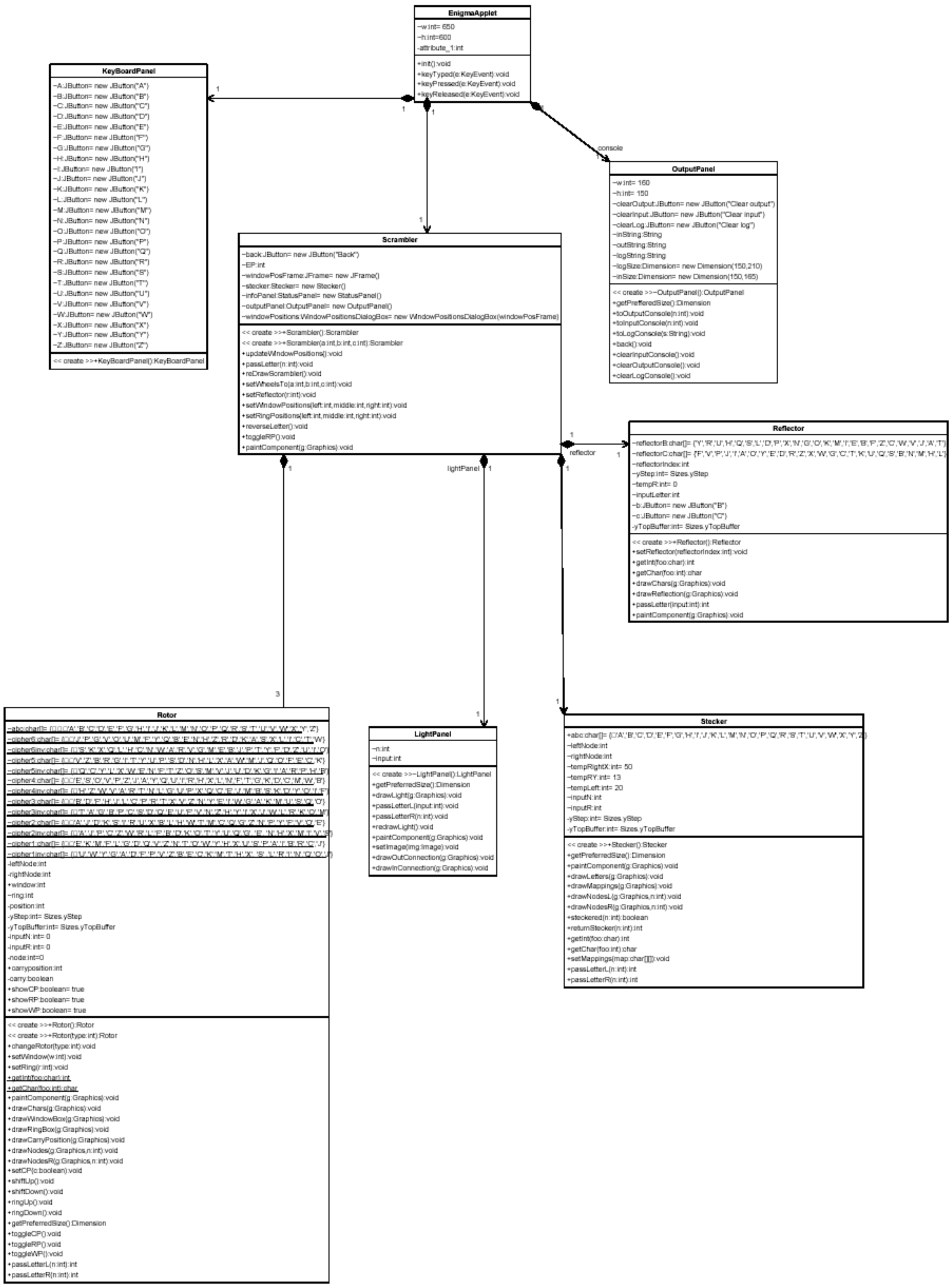

EnigmaApplet
-w:int= 650
-h:int=600
-attribute_1:int
+init():void
+keyTyped(e:KeyEvent):void
+keyPressed(e:KeyEvent):void
+keyReleased(e:KeyEvent):void
KeyBoardPanel
-A:JButton= new JButton("A")
-B:JButton= new JButton("B")
-C:JButton= new JButton("C")
-D:JButton= new JButton("D")
-E:JButton= new JButton("E")
-F:JButton= new JButton("F")
-G:JButton= new JButton("G")
-H:JButton= new JButton("H")
-I:JButton= new JButton("I")
-J:JButton= new JButton("J")
-K:JButton= new JButton("K")
-L:JButton= new JButton("L")
-M:JButton= new JButton("M")
-N:JButton= new JButton("N")
-O:JButton= new JButton("O")
-P:JButton= new JButton("P")
-Q:JButton= new JButton("Q")
-R:JButton= new JButton("R")
-S:JButton= new JButton("S")
-T:JButton= new JButton("T")
-U:JButton= new JButton("U")
-V:JButton= new JButton("V")
-W:JButton= new JButton("W")
-X:JButton= new JButton("X")
-Y:JButton= new JButton("Y")
-Z:JButton= new JButton("Z")
<< create >>+KeyBoardPanel():KeyBoardPanel
console
OutputPanel
-w:int= 160
-h:int= 150
-clearOutput:JButton= new JButton("Clear output")
-clearInput:JButton= new JButton("Clear input")
-clearLog:JButton= new JButton("Clear log")
-inString:String
-outString:String
-logString:String
-logSize:Dimension= new Dimension(150,210)
-inSize:Dimension= new Dimension(150,165)
<< create >>~OutputPanel():OutputPanel
+getPreferredSize():Dimension
+toOutputConsole(n:int):void
+toInputConsole(n:int):void
+toLogConsole(s:String):void
+back():void
+clearInputConsole():void
+clearOutputConsole():void
+clearLogConsole():void
Scrambler
-back:JButton= new JButton("Back")
-EP:int
-windowPosFrame:JFrame= new JFrame()
-stecker:Stecker= new Stecker()
-infoPanel:StatusPanel= new StatusPanel()
-outputPanel:OutputPanel= new OutputPanel()
-windowPositions:WindowPositionsDialogBox= new WindowPositionsDialogBox(windowPosFrame)
<< create >>+Scrambler():Scrambler
<< create >>+Scrambler(a:int,b:int,c:int):Scrambler
+updateWindowPositions():void
+passLetter(n:int):void
+reDrawScrambler():void
+setWheelsTo(a:int,b:int,c:int):void
+setReflector(r:int):void
+setWindowPositions(left:int,middle:int,right:int):void
+setRingPositions(left:int,middle:int,right:int):void
+reverseLetter():void
+toggleRP():void
+paintComponent(g:Graphics):void
reflector
lightPanel
Reflector
-reflectorB:char[]= {'Y','R','U','H','Q','S','L','D','P','X','N','G','O','K','M','I','E','B','F','Z','C','W','V','J','A','T'}
-reflectorC:char[]= {'F','V','P','J','I','A','O','Y','E','D','R','Z','X','W','G','C','T','K','U','Q','S','B','N','M','H','L'}
-reflectorIndex:int
-yStep:int= Sizes.yStep
-tempR:int= 0
-inputLetter:int
-b:JButton= new JButton("B")
-c:JButton= new JButton("C")
-yTopBuffer:int= Sizes.yTopBuffer
<< create >>+Reflector():Reflector
+setReflector(reflectorIndex:int):void
+getInt(foo:char):int
+getChar(foo:int):char
+drawChars(g:Graphics):void
+drawReflection(g:Graphics):void
+passLetter(input:int):int
+paintComponent(g:Graphics):void
Rotor
-abc:char[]= {'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'}
-cipher6:char[]
-cipher6inv:char[]
-cipher5:char[]
-cipher5inv:char[]
-cipher4:char[]
-cipher4inv:char[]
-cipher3:char[]
-cipher3inv:char[]
-cipher2:char[]
-cipher2inv:char[]
-cipher1:char[]
-cipher1inv:char[]
-leftNode:int
-rightNode:int
+window:int
-ring:int
-position:int
-yStep:int= Sizes.yStep
-yTopBuffer:int= Sizes.yTopBuffer
-inputN:int= 0
-inputR:int= 0
-node:int=0
+carryposition:int
-carry:boolean
+showCP:boolean= true
+showRP:boolean= true
+showWP:boolean= true
<< create >>+Rotor():Rotor
<< create >>+Rotor(type:int):Rotor
+changeRotor(type:int):void
+setWindow(w:int):void
+setRing(r:int):void
+getInt(foo:char):int
+getChar(foo:int):char
+paintComponent(g:Graphics):void
+drawChars(g:Graphics):void
+drawWindowBox(g:Graphics):void
+drawRingBox(g:Graphics):void
+drawCarryPosition(g:Graphics):void
+drawNodes(g:Graphics,n:int):void
+drawNodesR(g:Graphics,n:int):void
+setCP(c:boolean):void
+shiftUp():void
+shiftDown():void
+ringUp():void
+ringDown():void
+getPreferredSize():Dimension
+toggleCP():void
+toggleRP():void
+toggleWP():void
+passLetterL(n:int):int
+passLetterR(n:int):int
LightPanel
-n:int
-input:int
<< create >>~LightPanel():LightPanel
+getPreferredSize():Dimension
+drawLight(g:Graphics):void
+passLetterL(input:int):void
+passLetterR(n:int):void
+redrawLight():void
+paintComponent(g:Graphics):void
+setImage(img:Image):void
+drawOutConnection(g:Graphics):void
+drawInConnection(g:Graphics):void
Stecker
+abc:char[]= {'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'}
-leftNode:int
-rightNode:int
-tempRightX:int= 50
-tempRY:int= 13
-tempLeft:int= 20
-inputN:int
-inputR:int
-yStep:int= Sizes.yStep
-yTopBuffer:int= Sizes.yTopBuffer
<< create >>+Stecker():Stecker
+getPreferredSize():Dimension
+paintComponent(g:Graphics):void
+drawLetters(g:Graphics):void
+drawMappings(g:Graphics):void
+drawNodesL(g:Graphics,n:int):void
+drawNodesR(g:Graphics,n:int):void
+steckered(n:int):boolean
+returnStecker(n:int):int
+getInt(foo:char):int
+getChar(foo:int):char
+setMappings(map:char[][]):void
+passLetterL(n:int):int
+passLetterR(n:int):int

Figure 10. The progression of the model so far. (see Appendix pages for better image)

3.5.5. Steckerboard and Stecker Dialog Box

This was the first dialog box I used to control user-defined settings. Most of the Enigma machine and its corresponding components are simple mechanisms whose difficulty lies only in their understanding. Because of this I thought the steckerboard would be easy to implement as all it did was to cross/swap letters over so that if letter ‘A’ was steckered to ‘S’ then pressing ‘S’ would result in actual fact to ‘A’ being sent through the scrambler. And if ‘S’ was the result of a letter being entered into the scrambler the light of letter ‘A’ would be lit instead.

The first versions of the Enigma had no steckerboard and letters pressed reached the scramblers un-swapped. In the later version of 1938, the steckerboard was introduced and came with 6 (or 7) steckers multiplying the number of existing permutations. Later Enigmas came with with 10 steckers, which mathematically speaking is the optimal number in order to maximise the number of permutations. I refrained from providing 10 steckers because I found that with the Enigma simulators I tried, I rarely went beyond using two, and six would be ample to demonstrate the point of the steckers.

Feeling more confident with layout managers in general, I attempted to use the GridBagLayout to set out the dialog box I would use to extract stecker settings from the user. The idea I had was a simple set of 2x6 textFields which when correctly filled in, would join or stecker letters as pairs on the same row.

Problems that I did not take in to account were

1. The need to make sure reciprocated steckers were implemented automatically
2. Each textField must not have more than one letter entered i.e. entry [AB] is not valid in a single textField
3. A steckered letter can only exist in one row, so if A is steckered to C on one row, A cannot be steckerd to B say, on another row
4. Partially entered rows needed to be either discarded or dealt with accordingly

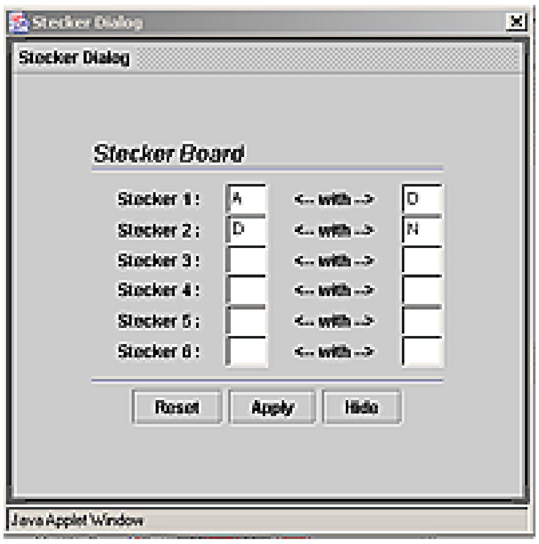
The requirement for textField not having more than one letter led to more requirements. For example, when entering stecker settings the user should not type in numbers or other non-alphabetical characters. Also, should there be a differentiation between upper-case and lower-case letters? Aesthetically, allowing both seemed visually detrimental yet allowing only upper-case would be too restrictive.

In the end I decided to extend the textField class in my own defined class

```
class LetterField extends TextField{
        char c;

   public LetterField()   {
                setColumns(1);
                addKeyListener(new KeyAdapter()       {
         public void keyTyped(KeyEvent e)
         {
                      char t;
                    t = e.getKeyChar();


//Allow only Letters to be typed and no more than 1, and backspace
if(((!(Character.isLetter(t))) || (getText().length() >0)) && (t !='\b'))
                e.consume();


        else{   //THIS PART TURNS ALL KEYPRESSES TO UPPERCASE
        c = Character.toUpperCase(t);
```

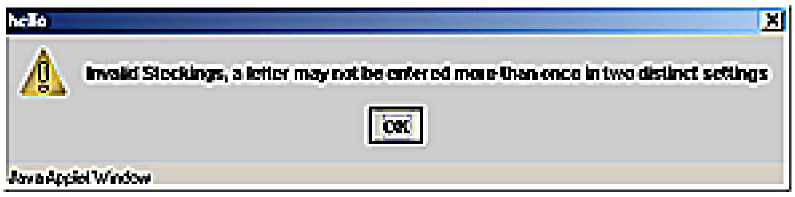
```
                e.setKeyChar(Character.toUpperCase(t));
                        }
                    }
                });
            }
        }
```

Using my own LetterField class derived from TextField I could override it ensure that only one character letter per textbox could be entered at anyone time, and letters typed would automatically be transposed to the uppercase equivalent. Any non-letter character other than backspace (“\n”) would be dissipated using the consume command.



Stecker 1 : A <-- with --> D
Stecker 2 : D <-- with --> N
Stecker 3 :
Stecker 4 :
Stecker 5 :
Stecker 6 :

Figure 11. Invalid steckings entered into the stecker dialog box



Invalid Steckings, a letter may not be entered more than once in two distinct settings

Figure 12. The warning dialog box used to highlight invalid settings

Another question that had to be asked was how to compare the stecker entries for double entry. I originally gave each of the letterFields individual names but realised that to compare 12 of them would require 12! individual equality tests. i.e.

```
(pseudo code):
        if ((stecker1= = stecker2)||(stecker1 = = stecker3)|| stecker1 = = stecker4)......
        ...so on up to (stecker11 = = stecker12)){
                generateSteckerWarningDialogBox();
        }
```

Instead of using names to identify each LetterField box, I used a 2x6 array of LetterFields. Thereby reducing the previous and extremely long comparison method to just a few lines of code that cycled over a set of nested for statements. Testing each LetterField[p][q] against all other LetterField[i][j] for equality making sure not to test a

LetterField against itself (!((p==i)&&(q==j))). The sample of code below is from the SteckerDialogBox class.

```
public boolean doubleCheck(){
        for(int p =0; p < 6; p++){
                for(int q =0; q < 2; q++){
                        for(int i =0; i < 6; i++){
                                for(int j=0; j < 2; j++){

                                        if (!((p==i)&&(q==j))){

                String temp1 = (arrayLF[p][q]).getText();
                String temp2 = (arrayLF[i][j]).getText();
                while ((temp1.length()>0) && (temp1.equals(temp2)) ){

                                        warning();

        //if a double entry of a letter exists then that pair is removed

                        (arrayLF[i][0]).setText("");
                (arrayLF[i][1]).setText("");
                        return false;
                }
                                        }
                                }
                        }
                }
        }
return true;
}
```

A problem I was experiencing before testing with the condition:

```
if ((temp1.length()>0) && (temp1.equals(temp2)) ){...
...}
```

If you failed to fill in any two LetterField boxes they would both be equal to null and therefore equal to each other raising the warning message box inappropriately. Worse still when trying to draw the results of the stecker dialog box to the stecker panel, a null value resulted in a node position drawn off the screen and a resulting line going off in an unknown tangent.

The plan was once the user had filled in the stecker dialog box and pressed submit, the entered steckers would be stored in a double char[2][6] array, which would then be passed as a parameter to the stecker panel to be drawn.

The stecker panel when passed a letter either from the right or left (entry/exit), would first test whether the letter passed existed in that double char array object. If so, then a second method would be called to return the alternative letter (the letter on the same row) to be used instead.

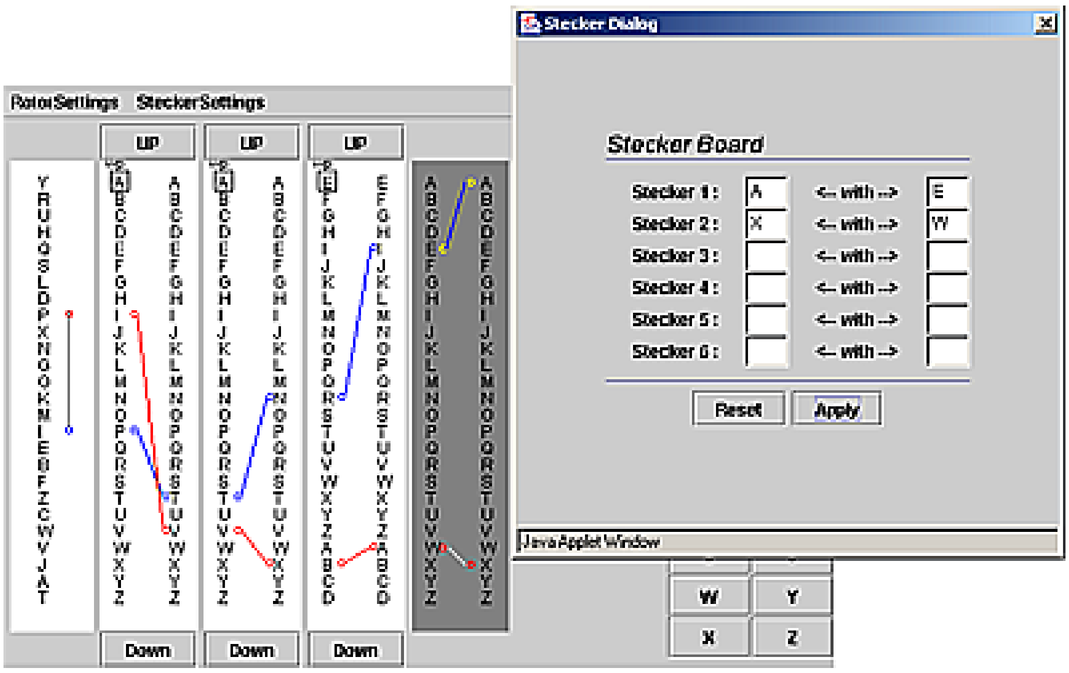
Using as an example, A is steckered to D and S is steckered with N :

```
(i  = 0):   [A] [D]
(i  = 1):   [S] [N]
```

if we have 'A' paired with 'D', and 'A' was passed, then getInt(map[i][1]) would be called, where i = 0 in this case. The character 'D' is the element at map[0][1] (the getInt() function returns the numerical position of that letter for convenience).
SImilarly, if 'D' was entered then getInt[map[i][0]) would be called where i = 0 and the numerical value of A would be returned.

```
public int returnStecker(int n){
        for(int i = 0; i < 6; i++){
                for(int j =0; j<2; j++){
                        if (getChar(n) == map[i][j]){
                                if(j ==0){
                                        return (getInt(map[i][1]));
                                }
                                else
                                        return (getInt(map[i][0]));
                                }
                }
        }
        return n;   //catch all statement
}
```

Once these worked satisfactorily, the two classes were combined together again in the Srambler class. And the desired result was confirmed, see Results section.



Stecker 1 :	A	<-- with -->	E
Stecker 2 :	X	<-- with -->	W
Stecker 3 :		<-- with -->	
Stecker 4 :		<-- with -->	
Stecker 5 :		<-- with -->	
Stecker 6 :		<-- with -->	

Figure 13. The stecker settings in the dialog box are applied to the scrambler

3.5.6. **Wheel Positions Dialog Box**

Continuing with the theme of dialog boxes and GridBagLayouts, I constructed a similar popup box to provide the user the option to change the placements of the wheels. Currently, the settings were hard coded in the order I, II, III, (left, middle, right or sometimes referred to as position1, position2, position3)

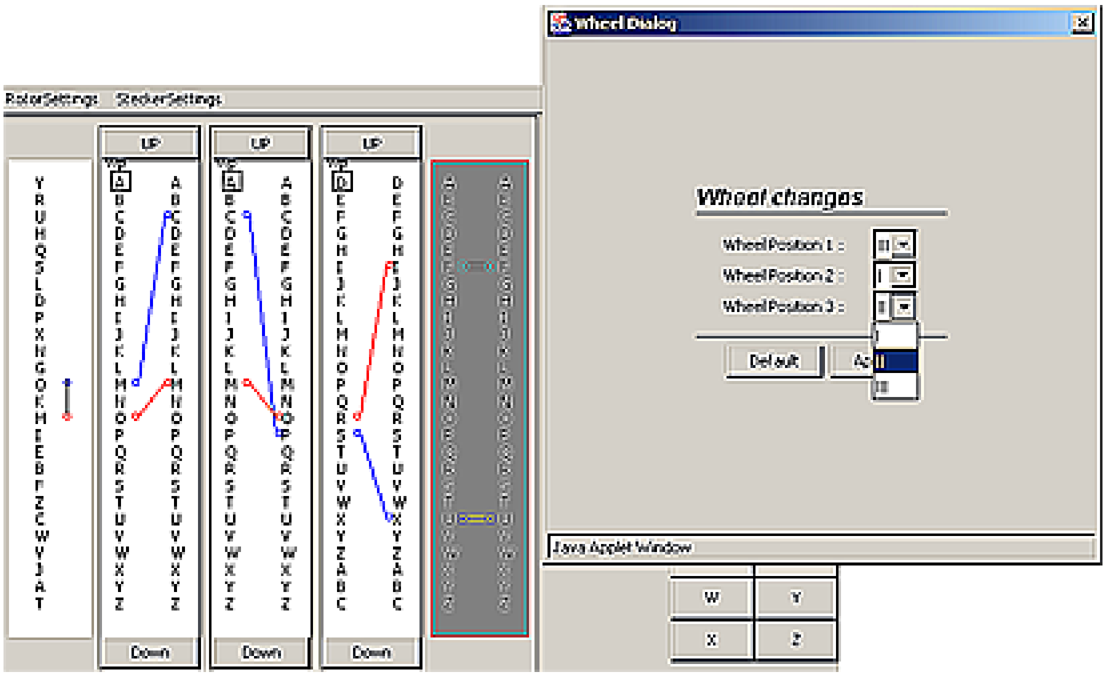


Figure 14. The wheel order can now be user defined

Instead of using LetterFields, it seemed intuitive to provide Combo boxes instead. There were a few problems which arose here, I was initially unable to change the rotors that I instantiated in the scrambler to different rotors. And after spending some time I found that even though I could extract the correct end result and that the calculations performed by the newly positioned rotors therefore worked, they would not display themselves properly, i.e. the changed rotor did not draw itself properly.
I can only guess that the newly positioned rotors were drawing themselves on a different Graphics context to the one being used. I think the context being used was still connected to the rotor that was previously in that position. Since the output of the scrambler suggests that the signals were correctly connected to the new rotor. The reason why the changed rotor did not draw any new changes was because that panel was still connected to the old rotor in that position which was no longer being used.

This problem was solved by redefining the mappings in that wheel to the mappings of the chosen wheel. To explain, instead of producing a new wheel with new settings and swapping them, I would just change the character mappings and turning positions to those of another wheel.

This is probably not the best way, but it seemed to work well. The only problem I could identify was that each rotor would still have the original rotorPosition ID, even after it had been changed. So even if rotor1 swapped with rotor3, it would still retain the identity of rotor1 even though it would now behave and pass letters like rotor3. This did not interfere with the way in which the machine worked, and if needed IDs could be changed easily.

The other problem was that of how I should display these changes in wheel positions, because unless you knew what had happened it could be interpreted as status quo. After some discussion with some test users I decided a status label was needed to reflect the states of all the variables that might change during the time of operation by the user.

Incidentally, I felt it not worthwhile to create another dialog to cater for the changes in reflector since it only existed in one position and could only be chosen from two. Because of this I later included it within the same dialog box as the wheel changes.

3.5.7. Window Settings and Ring Settings Dialog Panel

The layout of the ring-settings dialog box and the window-settings dialog box were very similar and in hindsight I was possibly guilty for not programming in a very object-orientated way

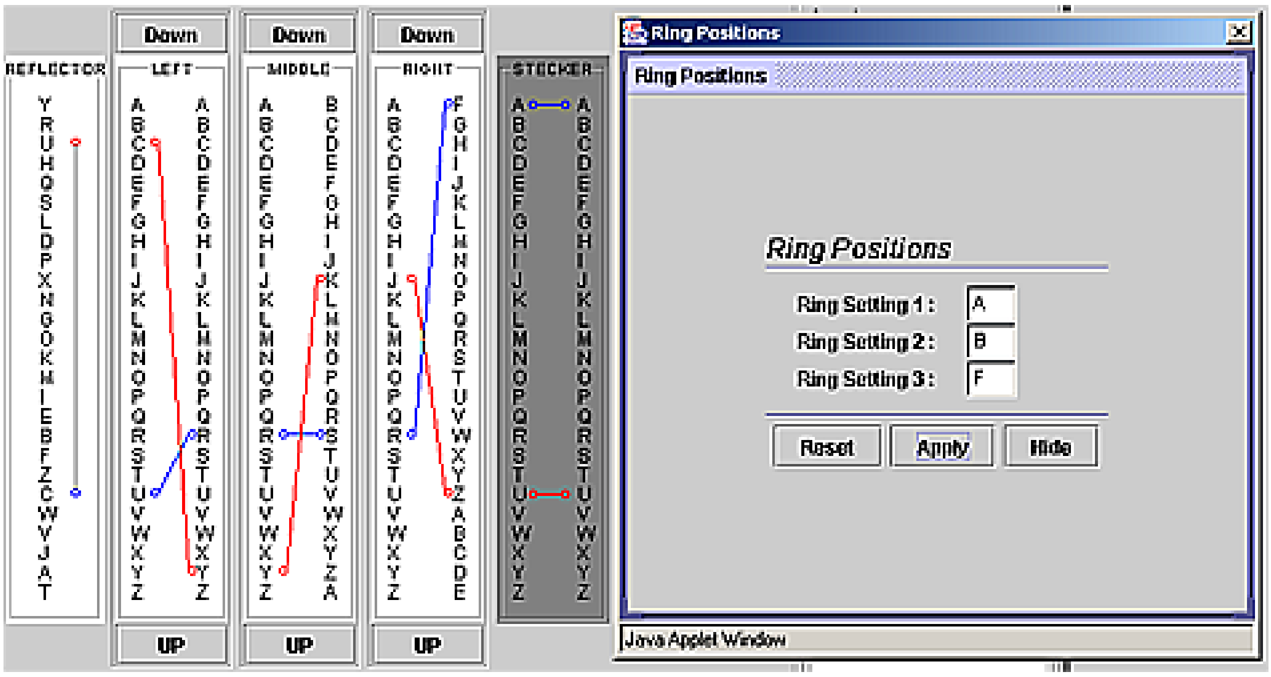
By this time, I still had not fully worked out exactly how I was to represent the ring-settings of a rotor and I could not defer it any longer.

Lots of mistakes were made during this process such as adding both the window positions and ring positions. But after analysing the mappings again, taking into account the ring positions properly, I finally ended up with the following functions that did give correct outputs.

```
public int passLetterL(int n){
        inputN = n;
        return ((getInt(rotor[(n+window-ring+26)%26])+26-window+ring)%26);
}
public int passLetterR(int n){
        inputR = n;
        return ((getInt(rotorInv[(n+window-ring+26)%26])+26-window+ring)%26);
}
```

The liberal use of adding 26 and modulising by the same number was needed to prevent array access below mapping elements 0 and above 25, which were sometimes caused by the addition and subtraction of the window and ring positions.

And because the ring-positions are simply relative positions, I decided I would show this by shifting the right side of the wheel.



Ring Positions

Ring Setting 1: A
Ring Setting 2: B
Ring Setting 3: F

Figure 15. The relative positions of the inner rings are reflected in the shifting of all the letters

Before, all rotors were default to a ring setting of AAA, so there was no shifting of the inner core resulting in each rotor being uniform in its two columns of letters (as with the left rotor above). However, a change in the ring position to 'B' will cause all the entry positions to shift by one so the entry position of 'A' is in fact now substituted by 'B', and 'B' substituted by 'C' and so on.
Likewise, if the ring position is 'F' as with the right hand rotor, then there will be a relative shift of five positions.

When it came to implementing a similar dialog box for window settings, I decided I wanted the dialog box to automatically reflect the state the window positions at the point the dialog box opened. This requirement was different to previous dialog boxes like the ring setting dialog box, because the ring settings could only be changed using that dialog box, and once changed they would remain in that state until the next time the user opened up the ring settings dialog box. For example, the above ring Settings ABF would remain ABF until the user reopened that dialog box and changed them, whereas the window positions are changed every time a letter signal is passed through the scrambler or when the user changes them via the up and down buttons.

I did this by writing a method within the window positions dialog box and calling the method any time the scrambler passed a letter or shifted any of its rotors.

```
public void updatePositions(int A, int B, int C){
        window1.setText(String.valueOf(Rotor.getChar(A)));
        window2.setText(String.valueOf(Rotor.getChar(B)));
        window3.setText(String.valueOf(Rotor.getChar(C)));
}
```

3.5.8. Keyboard

A simple keyboard was reproduced using three panels for each tier of the keyboard. The three tiers would each have a FlowLayout, and with the middle row having one less key would align itself to centre and create an the offset look keyboards have.
I tried using various layout managers, but finally settled for this. I did want to use graphical buttons, but considering how much space the rotors already took I could not really afford the space that a set of realistic round buttons would take. I question how much a realistic set of buttons would add to the simulator at any rate especially since a completely authentic looking Enigma was no longer my chief aim.

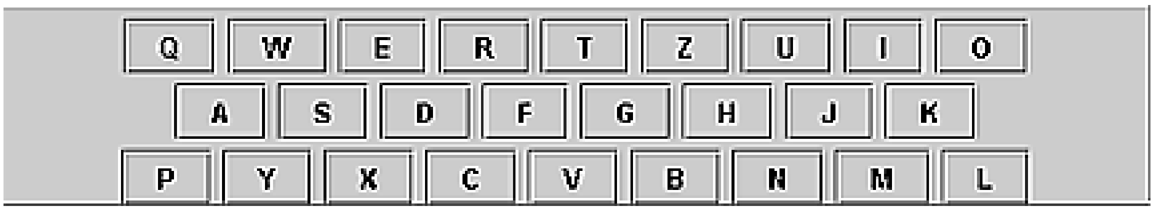


Figure 16. Keyboard Panel, made up of three tiers of JPanels, each tier using a flow layout

However, I did make up for lack of authenticity by implementing the keyListener interface to allow input via the user's computer keyboard using the corresponding JButton's doClick() method. Though, through doing this I ran into a new set of problems relating to the focus of the applet.
The keyListener interface was registered at the top level of the applet class, since this was the only way I could get the listener to work, but once a button was clicked using the mouse the focus of the applet was lost and the computer's keyboard would no longer work, (not in the lifetime of the applet).

It took a while to just figure out why this was happening, I thought about removing the feature but felt it added a lot the simulator by not forcing the user to only interact with the Enigma via the keyboard buttons I drew on screen.

I finally got around this problem by including a button within the top layer applet class that performed the function of regaining focus (for the applet). The button would not be visible since it would not be added to the user interface, but it could be actioned in the keyboard class by using the doClick() method every time a button was clicked. Thereby always returning focus to the applet and the keyboard.

3.5.9. Light Bulb Panel

The light panel was used to display the resulting output of passing a letter through the scrambler. I first tried drawing a light bulb using the fillPolygon() method, but found this troublesome. Instead I used a graphics package to create an image of a light bulb which I would place directly behind the letter I wanted to highlight it as being the resultant output. The first light bulb I created had a grey base which did not show up well against the grey panel and various colours were used until I found one that I was happy with.

Seeing as I had drawn a path from the scrambler to the resulting light bulb, I took the opportunity to draw a path from the keyboard to the scrambler to complete the path taken from keyboard to scrambler and scrambler to light bulb.

3.5.10. Output Panels

An idea suggested by my supervisor was that I could give also give a text description of what was going on each time a key was pressed. This was easily implemented since such a description already existed and sent to the console to aid my search for bugs.
The problem lay in where I could place the output since space vertically was already tight. I still needed to create an output panel and input panel so that the two could be compared by the user, and planned to place these on the East side of my applet class so not disturb the alignment between the scrambler and the keyboard. If I were to add an extra panel to communicate the working of the machine, it would have to be included with the input and output panels.

The text for the output and input panels were changed to the font MonoSpaced, so that their sizes would be equal. This made it easier to align the characters in blocks of 4 or 5 characters and to compare the input and output panels against each other.

3.5.11. Sound

From an site devoted to rebuilding Enigmas (Enigma Replicas), I managed to download an actual sound recording of an enigma's action as a key was pressed. Using the getAudioClip method inherited from the Applet class in a similar way in which I used the getImage method earlier to load an image file.

```
AudioClip clunk = getAudioClip(getCodeBase(), "keyclunk.wav");
```

I was able to load my sound to the applet and place a method clunk.play() in the body of Scrambler's passLetter() method so that the sound clip would be played each time a letter was passed.

3.5.12. Back Option

One feature I felt the simulator needed, (after I had added the keyListener to recognise physical keyboard presses) was the need for a backspace function or delete key.
This would allow the user to remove one character from each of the ouput and input consoles. Thinking slighly beyond this made me think that the wheels should rotate back one shift so that the machine would be in exactly the same state as it was before the wrongly entered character was typed.

This was easy to implement since it would be a case of reducing the input and output consoles by a character each time and turning the right rotor back the same number of times. But what if a turn I was trying to reverse had made another rotor(s) turn?
This was something I needed to be careful about. After looking at the problem more closely I soon found out it was not at all difficult, the turning positions going back worked in exactly the same way turning positions had been programmed to go forward in the scrambler class. So, instead of shifting up each time we have,

```
//Scrambler reverses motion
        public void reverseLetter(){
                rotorPos3.getRotor().shiftDown();
                updateWindowPositions();


                if (rotorPos3.getRotor().window == rotorPos3.getRotor().carryposition){
                                rotorPos2.getRotor().shiftDown();

                //nested second if statement
                if (rotorPos2.getRotor().window == rotorPos2.getRotor().carryposition){


                                rotorPos1.getRotor().shiftDown();
                                }
                        }
        outputPanel.back();
        reDrawScrambler();

        }
```

Not wanting to mess around with the layout of the keyboard I was already happy with, I added the function in the tool bar and registered the KeyEvent_VK_DELETE to perform the action also.

3.5.13. Look & Feel

Much of the look and feel of the simulator was added through trial and error. I wanted to put borders around all of the panels I had used and went through all of the available borders in the Swing classes before settling on a combination of a TitledBorder, encased by a simple EtchedBorder where I felt identification was need, and just an EtchedBorder everywhere else.

The other change I implemented in the design was the introduction of the setLookAndFeel method, which allows the designer to determine the style in the program is displayed.

```
        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
        javax.swing.SwingUtilities.updateComponentTreeUI(getContentPane());
        javax.swing.SwingUtilities.updateComponentTreeUI(d);
        javax.swing.SwingUtilities.updateComponentTreeUI(wp);
        javax.swing.SwingUtilities.updateComponentTreeUI(ringPositions);
        javax.swing.SwingUtilities.updateComponentTreeUI(keyrotor);
```

Rather than just making the applet draw itself in the style of one window type, I decided to give the option to the user and place the option in the menu bar. The method `updateComponentTreeUI`, updates all elements tree elements on the component. It seems my Dialog boxes are not a part of that definition in regards to the JApplet class itself and need to be updated separately.

3.5.14. Status Bar

These were a set of 6 Labels, embedded in a narrow JPanel class, placed in the scrambler where adding a method to the listeners and within existing methods would ensure that the correct labels were updated whenever the relevant components changed.

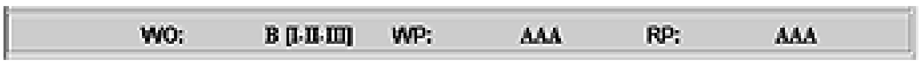


Figure 17: Status bar showing current wheel order, window position and ring position

3.5.15. Tooltips

This was the last element added to the Enigma as an afterthought when I need d to provide more information about some components but was confronted with a lack of space to do so. For each component that required a tool-tip, it was a simple exercise to add

```
MyComponent.setToolTipText("Component Information Message");
```

4. Results

The definitive test of course would be to compare the simulator against an actual Enigma, but part of the reason why I am building a simulator is because there are so few of these machines left in a working state. My alternative where necessary was to compare my output with other Enigma simulators.

4.1. Test 1 – Scrambler Output

Theoretically, if the actions of my simulator are correct and I used the same rotor mappings provided as the simulator I was comparing to (Carlson), then I should be able to reproduce trace results similar to the ones he so helpfully provided

The trace description given by Carlson:

```
ABDC CCDD DDFF F->S SSSS SSEE EFCB <<<<
BDHF FFII IIVV V->W WWNN NNTT TVLJ
CFLI IIXX XXRR R->B BBWW WWMM MPHE
DHPL LLHH HHQQ Q->E EEAA AAAA AEPL
EJTO OOMM MMOO O->M MMCC CCPP PUWR
FLVP PPCC CCMM M->O OOMM MMOO OUWQ
GNNG GGRR RRUU U->C CCYY YYVV VCGZ
```

I’ve taken the liberty to highlight parts of his output to aid my brief description:
The first line in bold is the trace output given when A is entered into the scrambler
The 1st letter that goes through a rotor is underlined.

I wrote a simple Wheel class (see appendix) and small program to test the output of passing the Letter ‘A’ through the above settings.

The trace output of a simple rotor application I wrote to emulate the results (Appendix: RotorTest1.java)

```
        T1-A    T2-B    WP = 1  T3-D    output-C
        Exited from 1 is : C
        T1-C    T2-D    T3-D    output-D
        Exited from 2 is : D
        T1-D    T2-F    T3-F    output-F
        REFLECTOR PLATE RECEIVES-------------> F
 ---->
        T1-S    T2-S    T3-S    output-S
        Exited from 4 S
        T1-S    T2-E    T3-E    output-E
        Exited from 5 E
        T1-E    T2-C    T3-C    output-B
          ANSWER/OUTPUT IS -------------------> B
```

The result of my trace output seemed in the main part agree with Andy Carlson’s.

Though the format of the results look very different I have made bold and underlined the respective parts for comparison. The important parts to look for when comparing the two traces, are the letters sent into a rotor, the resultant letter exited from a rotor, the value before and after the letter goes through the reflector, the trace coming back and the final letter produced.

The result above shows only that of passing the letter A through the three rotors and scramble, but passing the letters ABCDEFG into my program did produce the required output BJELRQZ, and a full set of my results are included as part of the appendix(1c).

The previous test served only as a preliminary study of the rotor's mapping, and I tried a similar test using a version, which drew the results graphically.

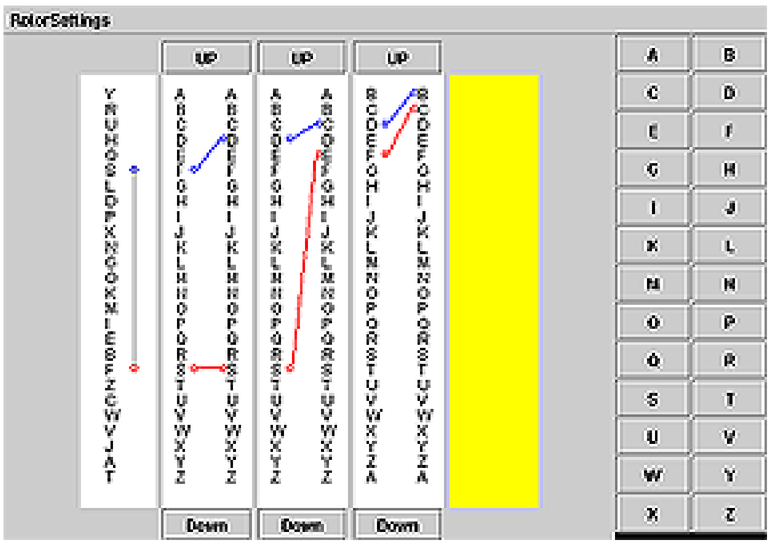


Figure 18. Graphical Implementation of Trace Results

Test 1 passed.

4.2. Test 2 - Test the wheel turnover position

Each Enigma's rotor had a turning position, which after a set number of rotations made the adjacent rotor on its left turn also.
With my current set up, rotor I on the left, rotor II in the middle and rotor III on the right (I,II,III). Rotor III turns on every key press but when it reaches its turning point, which for Rotor III is the letter 'W', the rotor on the left should turn one position.
Therefore, at window positions AAV any letter passed will turn the right rotor which in turn should turn the middle rotor. Resulting in window positions: ABW.

Similarly, Rotor II's turning position is the letter 'E', and at window positions ADV, passing a letter will turn all three rotors and the resulting window positions should be BEW.

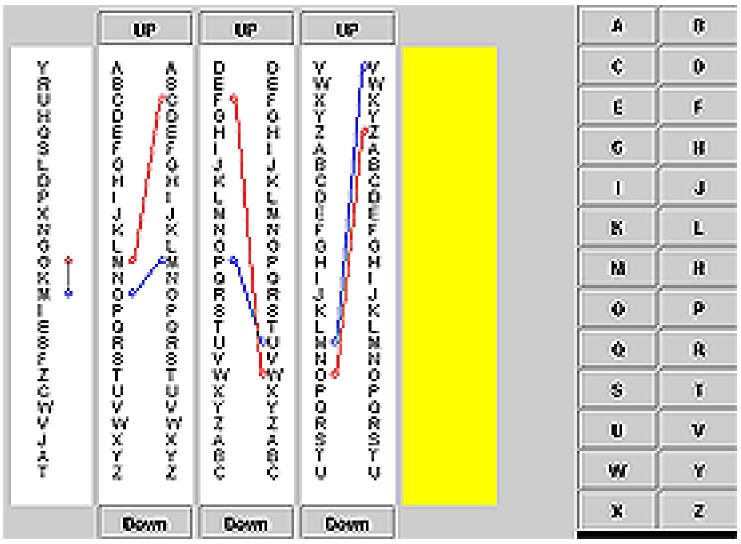


Figure 19. Current Window Positions: ADV (top row)

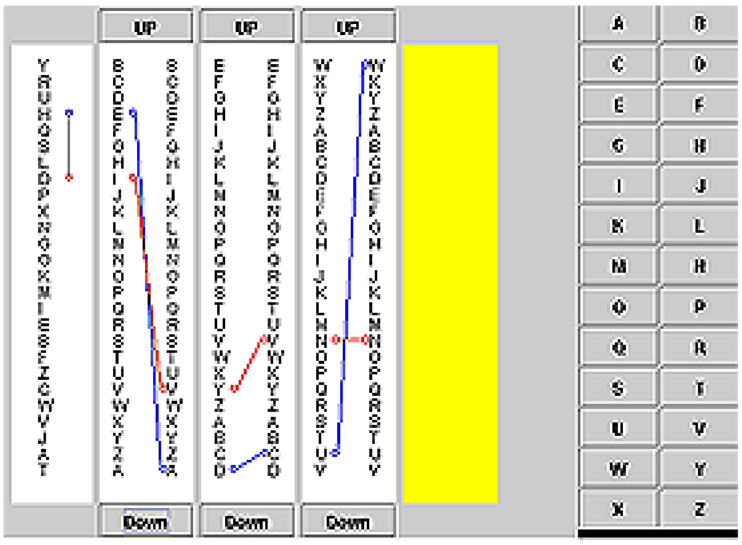


Figure 20. New Window Positions After Key Pressed: BEW (top row)

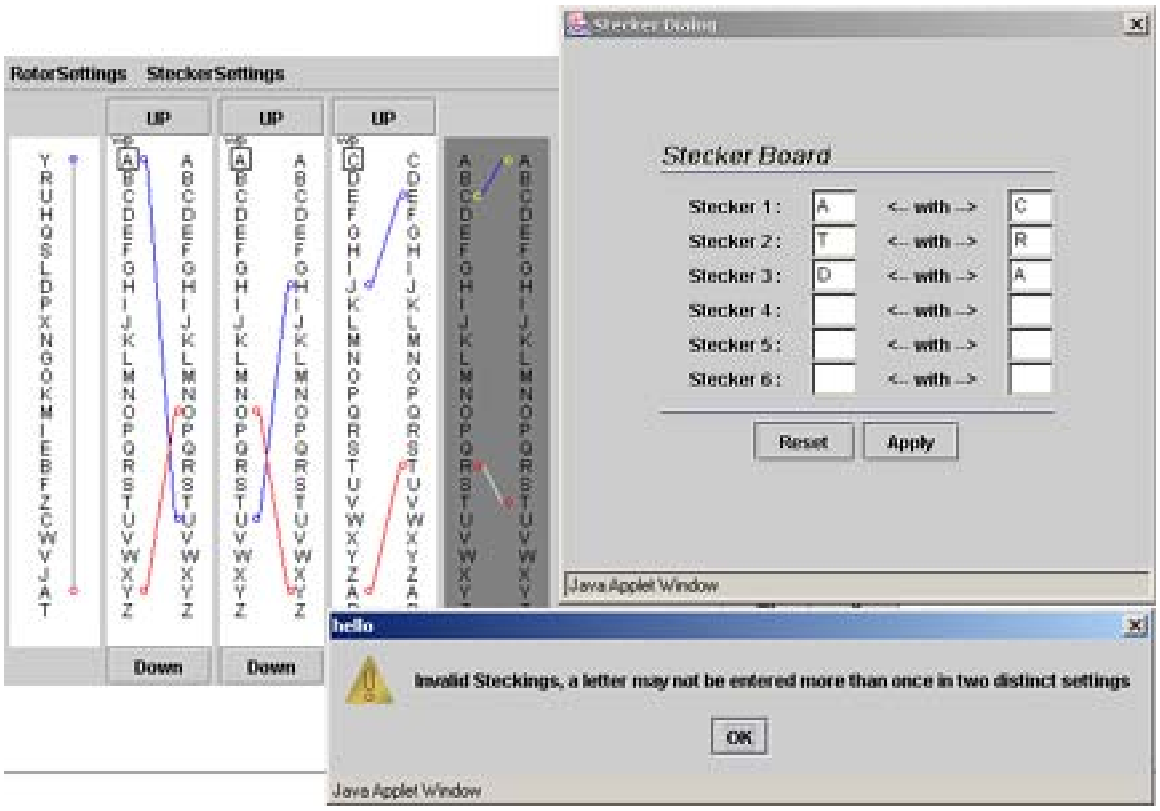
Test 2 passed.

4.3. Test 3 - The Stecker

The yellow panel, in the last Figure, showed where the stecker component was going to be. There were many parts to this problem that I had not foreseen until I started to write the Stecker class. For example, a set of steckers cannot be connected to a letter more than once, therefore error prevention needed to be included so to prevent the user from entering illegal settings. I did this by using a warning Dialog Box that preventing the user from continuing until legal stecker settings were entered.

But the primary aim, objective and therefore the measure of how successful my stecker was could be defined as the following: Once a valid set of steckings are entered the Scrambler should swap any steckered letters that occur on entry or exit of the Scrambler. Needless to say any if there are any occurrences, then the steckered letter should be used instead and drawn to screen.

Test 3 passed.



Stecker 1 : A <-- with --> C
Stecker 2 : T <-- with --> R
Stecker 3 : D <-- with --> A
Stecker 4 : <-- with -->
Stecker 5 : <-- with -->
Stecker 6 : <-- with -->

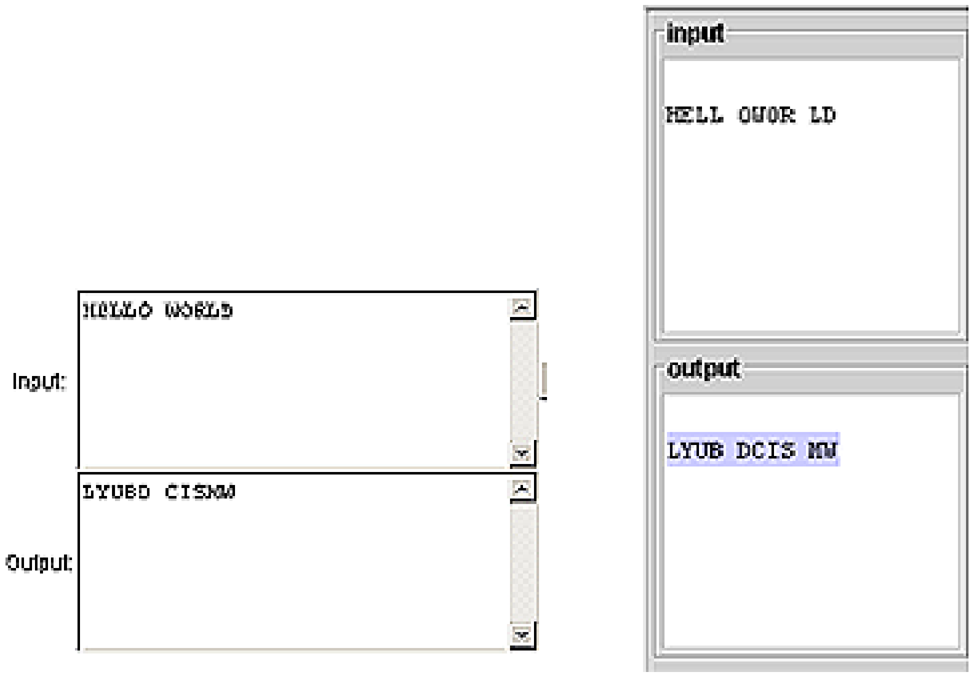
Invalid Steckings, a letter may not be entered more than once in two distinct settings

Figure 21. Applied steckers, swapping the entry and exit through the scrambler

4.4. Test 4 - The Ring Position

There was no text description to which I could compare to for this part, so I had compare results with Andy Carlson's Enigma simulator.

Using a wheel order of I,II,III, ring settings XYZ, window settings DOA and no steckers, my simulator encoded the message HELL OWOR LD to LYUB DCIS MW. Performing the same exercise with Andy Carlson's Enigma using those same settings yielded the same message.



```
HELLO WORLD
LYUBD CISMW
```

```
HELL OWOR LD
LYUB DCIS MW
```

Figure 22. Helloworld using Carlson's simulator (left) and mine (right)

Test 4 passed.

4.5. Test 5 - Rotor Positions

Again I have no option but to test this feature against Andy Carlson's Enigma.
So choosing a random set of settings:

Wheel Order	C - IV, I, III
Window Positions	JKW
Ring Positions	ADT
Steckers	A←→V, R← →Z

My simulator encoded `HELL OWOR LD   as   IFFB ZLHX CC`
Carlsons simulator `:HELLO WORLD    as   IFFBZ LHXCC`

Test 5 passed.

4.6. Go_Back/Delete

The purpose of the delete option was to remove the last letter typed from the output and input panels. After thinking about it a little, I decided it would be better if the positions of the Enigma (only the window positions should have changed), could be turned back to the previous setting as well (prior to the key being pressed).
To test this my Enigma performed this action I would set up the window positions so that at least one rotor other than the right rotor would turn during the typing of my message. Enter the message. Then use the go_back (in the menu bar under, clear console) or delete key. Going back the same number of times as a key was pressed should take the user back to the same settings as when the message was encoded.

Using wheel positions B-IV, III, I
window positions JAO
ring positions AAA
no steckers

Typing the string GORD ONWE LCHM AN – produces XFSY JWBY TSQS GQ, results in the window settings are now JBC. Pressing delete, or going_back 14 times (the length of the message) should

1. Take the user back to the original settings i.e. just the window settings JAO.
2. The input console and output console should be cleared.

The first criteria, which is the most important part works and therefore satisfies the test. But the second fails, by leaving three characters in each console box. The reason why this is so is because it regards the white space that was added to aid user's visibility as a character. With three white space characters in the above message, this seems the most likely cause of why three characters remain.

Test – Failed 22/9/03
Passed 23/9/03
(Eventually solved by removing two spaces if the last character = ' ')

4.7. User Feed Back

I needed to find out how user friendly the simulator was, did it achieve my aim of aiding the understanding of what functions the Enigma performed.

It seems that the concept of the Enigma is not an easy one to grasp, and just subjecting a user to the simulator in the hope he or she will understand the workings is just not feasible.

I therefore prepared a set of instructions (see appendix) and used those to brief my test user with an overview of what the Enigma was and how my simulator related to this. After leaving the user alone with the simulator for15 minutes, I then conducted an informal discussion to elicit information about the simulator.

One of the points that kept arising, was that it wasn't totally obvious what settings could be changed by the user, and when they were changed, their new state could not be observed. I later added a status bar to address this problem.

The biggest problem was that some people still had problems understanding what was going on despite the user manual. It seems whilst some Enigmas didn't provide enough information, my Enigma provided possibly too much information.

Also nobody I had asked felt that the simulator any resembling look of good quality. This is something I tried to address using the LookAndFeel option provided in the Swing package.

4.8. Test a final message

Randomly selecting the following Enigma settings.

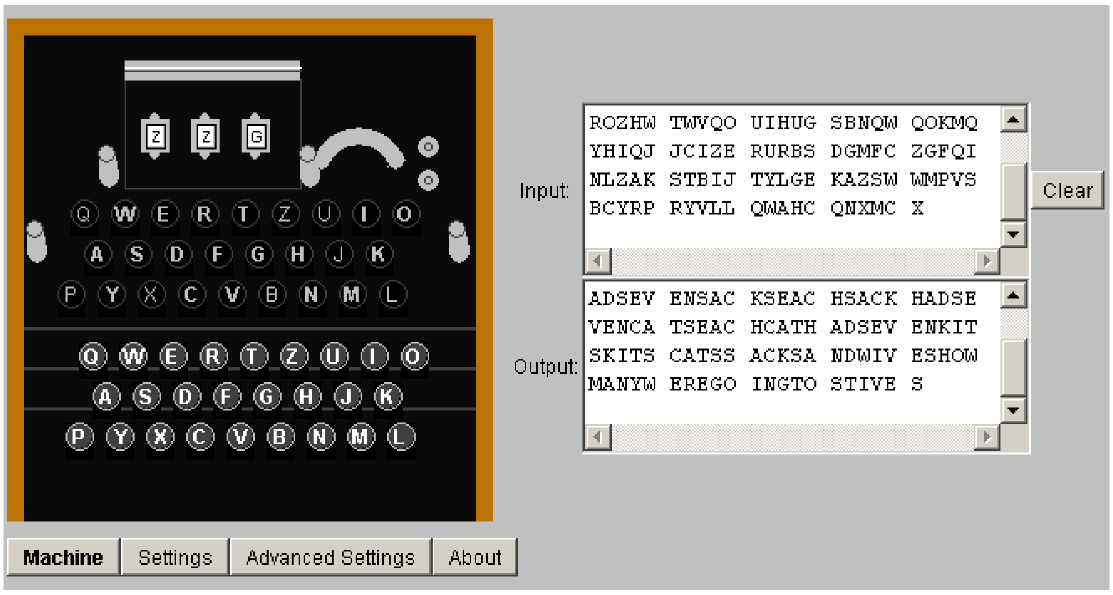
WO	[III, I, IV]
WP	ZUQ
RP	WQD

Steckers P↔W, S↔Q, O↔A, Z↔M, B↔C

Using these settings on my simulator and testing a fairly sentence (left) gives the encrypted string (right)

ASIW ASGO INGT OSTI	VUCY YLJZ AVUN NKVZ
VESI META MANW ITHS	DNWF EBGC HUUE DFBJ
EVEN WIVE SEAC HWIF	FFWE LBWG UFXM MKUM
EHAD SEVE NSAC KSEA	WSRO ZHWT WVQO UIHU
CHSA CKHA DSEV ENCA	GSBN QWQO KMQY HIQJ
TSEA CHCA THAD SEVE	JCIZ ERUR BSDG MFCZ
NKIT SKIT SCAT SSAC	GFQI NLZA KSTB IJTY
KSAN DWIV ESHO WMAN	LGEK AZSW WMPV SBCY
YWER EGOI NGTO STIV	RPRY VLLQ WAHC QNXM
ES	CX

And using that same encrypted string and entering it into Carlson's Enigma with the same settings produces:



Input:

```
ROZHW TWVQO UIHUG SBNQW QOKMQ
YHIQJ JCIZE RURBS DGMFC ZGFQI
NLZAK STBIJ TYLGE KAZSW WMPVS
BCYRP RYVLL QWAHC QNXMC X
```

Output:

```
ADSEV ENSAC KSEAC HSACK HADSE
VENCA TSEAC HCATH ADSEV ENKIT
SKITS CATSS ACKSA NDWIV ESHOW
MANYW EREGO INGTO STIVE S
```

Figure 23. Same settings, different machine

There are over 300 billion, billion start settings and in this chapter I have only looked at a handful of examples. But it seems that the results are both

- consistent with itself because the actions of the simulator are correct
- consistent with Carlson's simulator because the mappings for the rotors are the set the same.

Overall I would consider the tests were all completed successfully, I'm completely certain that the machine is 100% consistent with itself (i.e. you can code and decode using the same simulator), and I'm almost as sure that it is consistent with Andy Carlson's simulator as demonstrated above since I have used the same wheel mappings. With so many possible start settings it's impossible to be sure of course.

5. Conclusion

Looking back at my original project proposal, they were defined in two parts. The first was to build a fully working Enigma simulator, which behaved consistently with the descriptions written about, and secondly to investigate the techniques used to break the Enigma, ultimately to build a Turing Bombe.

I feel I more than successfully achieved this part. The simulator is both accurate and explains the inner workings of an Enigma machine. The use of flatly laid out rotors, goes further than currently existing Enigma simulators by clearly illustrating the path a signal takes depending upon the settings of the machine, and the resulting end result.

From the investigation that was conducted as part of the literature review, and other tests conducted along the way it does seem that my Enigma simulator is consistent with the descriptions described. The order of the rotor rotations, the mapping of the letters in-between wheels and inside the wheels, the Stecker-board and reflectors have all been written to perform the respective actions of those components in a seemingly correct way.

The bulk of my investigation on how Enigma messages were broken lies in the literature review chapter. To break any Enigma message required the code breaker to possess the machine settings. You are probably asking what kind of solution is that? But it seems there were a large number of ways in which one could and did get those machine settings, ranging from the primitive covert stealing of books of Enigma settings from German U-Boats to the exploitation in the machines weaknesses and intricacies from transmitted messages alone.

Obviously stealing the books worked well, but this only worked for very short periods (they were changed regularly) and when books were stolen without the enemy knowing - this was done very occasionally during the war. Without such help, any other solution boiled down to reducing the possible machine settings through educated guesses made from operator habits, machine inabilities and number theory. From the reduced number of machine settings these would be tested against an Enigma to see if a message entered into an Enigma came out as the expected message.

I was unable to completely satisfy my objectives for the second part, which were to conduct an investigation into the techniques and to build a device similar to the Bombe.

Despite work related commitments throughout the project, I do not feel these were the contributory factor in the second part's failings as I was well aware of my time constraints before choosing the project. Instead I cite the lack of clear understanding and information into the achievability of those objectives during the period in which the project was hastily chosen and bolted on.

It is difficult to predict how long it would take to construct a Bombe without first knowing how the Bombe works. To know how a Bombe works requires working knowledge of an Enigma, which I did not possess at the beginning of the project. (a good pinch of Number Theory would have also been useful). Without knowing how long the it would take I think I was justified to including the Bombe simulator only as a secondary objective.

5.1. Lessons Learnt from this Project

From a personal perspective, I achieved my wish to understand the Java Foundation Classes better by building what I consider a professional looking application/applet that explains the Enigma machine in detailed way.

I now have a greater knowledge and appreciation of the JFC limitations whilst gaining knowledge in subtle techniques used to make those classes behave in a way that I want. I give the example of overriding the getPreferredSize() technique used to control the sizes of components when the usual setSize() method doesn't work, the redefining of a TextField to only accept certain characters and the attention to component focus.

What I have also learnt is the importance of user testing as locking myself away and writing a user interface that I though was highly usable, was not an opinion shared by every other user.
Having recognised early on the fundamental need to model the rotors correctly I am guilty of not considering usability early enough. Looking at my design, I really do not know how I could make the system clearer, but what I should have done was to at least design some paper-prototype interfaces and investigate alternatives and thereby justifying my end product.

On reflection, other than not being too hurried/ambitious/naïve with my project proposal and objectives and the points just mentioned, I do not believe there is anything I would have done differently. I do not think it was feasible to draw too detailed a design plan so early on without first building the key components such as the Rotor class. Continual iteration and incrimination between cycles resulting in prototypes of class components which were either discarded or improved upon worked well for me without the burden of being tied down to plans that were drawn too early on and did not take into account difficulties in implementation.

Also I should mention the general ease and therefore surprise in which the final program was rid of bugs. I believe this was down to clear idea in which the classes were designed. I found that most problems discovered, such as if the reflector was not drawing correctly or if values were being entered into the next rotor wrongly, it was relatively straightforward to investigate the problem by knowing where to look. It helped of course that most of these classes represented physical components that I had studied and therefore a mental map already exists to some extent. And with this reasoning I would say that this has been a very rewarding exercise in learning how to program in an Object Orientated way.

5.2. Where the project could go from here

The obvious place would be to build a Polish Bomba, before tackling the Turing Bombe or maybe a simulator that demonstrates the Banburismus method clearly. Alternatively there were many variations and improvements to both the Enigma and Bombes, some such as the Colossus marked very important stages in the life of modern day computers, these would be interesting machines to simulate.

As was the case with the Enigma simulator, any of these would be an interesting and rewarding project to undertake. But to simulate something like Bombe first requires a good understanding of the Enigma. When I started this project I was lacking the necessary understanding but what I do hope is that the simulator I present here goes some way in helping someone else achieve that goal.

References

Books

Hugh Sebag-Montefiore, Enigma - *The Battle for the Code*, Phoenix paperback 2001

Andrew Hodges, *Alan Turing the Enigma*, Vintage Press 1983.

Journal Articles

The Bletchley Park translated Enigma Instruction Manual, transcribed and formatted by Tony Sale (c) 2001, downloaded (July 2003) from [http://www.codesandciphers.org.uk/documents/egenproc/egenproc.pdf]

David H. Hamer, Enigma: Actions Involved in the 'Double Stepping' of the Middle Rotor", Cryptologia XXI(1), January 1997

Weselkowski, Slawo. "The Invention of the Enigma and How the Polish Broke It Before the Start of WWII". http://www.ieee.org/organizations/history_center/cht_papers/wesolkowski.pdf

PRO-HW 25/3, Public Records Office, Turing's hand written notes made at Bletchley, October 2000.

Marian Rejewski, (1980) 'An application of the Theory of Permutations in Breaking the Enigma Cipher', Warsaw University.

Web addresses

Enigma and the Turing Bombe, Nik Shaylor, [http://frode.home.cern.ch/frode/crypto/Shaylor/bombe.html] (25 July 2003).

Turing's Treatise on Enigma, released 1999 reformatted by Frode Weirund, available to download from [*http://frode.home.cern.ch/frode/crypto/Turing/turchap1.pdf*] (25 July 2003).

Tony Sale, Bletchley Park homepage [http://www.codesandciphers.org.uk/index.htm] (25 July 2003).

US 6812 Bombe Report 1944, formatted by Tony Sale© 2002.
available to downloaded from
[http://www.codesandciphers.org.uk/documents/bmbrpt/usbmbrpt.pdf] (25 July 2003).

Graham Ellsbury 1998, The Turing Bombe
[http://www.ellsbury.com/bombe1.htm](26 July 2003).

Andy Carlson, Enigma Applet
[http://homepages.tesco.net/~andycarlson/enigma/enigma_j.html](23 August 2003)

Russell Schwarzger, Enigma Applet
[http://www.ugrad.cs.jhu.edu/~russell/classes/enigma/enigma.html](23 August 2003)

Enigma Replicas
[http://www.enigma-replica.com/index.html] (12 Sept 2003)

Bibliography

- Jim Arlow and Ila Neustadt, *UML and the Unified Process*(2002)*,* Addison Wesley.
- Charles Richter, *Designing Flexible Object-Orientated Systems with UML*, Pearson Education.
- Litwak, *Pure Java 2*(2000), SAMS publishing
- Geary, *Graphic Java – Mastering the JFC -AWT* (1999), Prentice Hall
- Geary, *Graphic Java – Mastering the JFC -Swing* (1999), Prentice Hall
- Chan, *The Java Developers Handbook(2000),* Sun Microsystmes
- Ken Arnold, James Gosling, *The Java Programming Language Second Edition*(1998) Addison Wesley.

Web sites

- The Cryptographic Mathematics of ENIGMA http://ed-thelen.org/comp-hist/NSA-Comb.html
- The US 6812th Division Report on the British Bombe, 1944 *http://www.codesandciphers.org.uk/documents/bmbrpt/index.htm*
- NOVA online http://www.pbs.org/wgbh/nova/decoding/enigma2.html
- The Public Records Office http://www.pro.gov.uk/
- The Alan Turing Home Page http://www.turing.org.uk/turing/
- Experts Exchange *http://www.experts-exchange.com/Programming/Programming_Languages/Java/*
- Google – Java Newsgroups http://groups.google.co.uk/groups?hl=en&lr=&ie=UTF-8&group=comp.lang.java

Appendices