[](http://www.google.co.uk/url?sa=i&rct=j&q=pates+grammar&source=images&cd=&cad=rja&uact=8&ved=&url=http://www.schooltogs.com/schooltogs&ei=JaI3VYXJBYrhaIyNgYAC&psig=AFQjCNFVEgg6wkOcJvB9BV_eWuOJO_YOEg&ust=1429795749580256)



**PATE’S GRAMMAR SCHOOL**

**COMPUTING DEPARTMENT**

**Unit 3/4 – Programming Project**

CANDIDATE NAME

Alex Daniels

EXAM NUMBER

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[(1) Anlysis of the problem (10 Marks) 3](#_Toc454442805)

[(i) problem definition 3](#_Toc454442806)

[(ii) Stakeholders 5](#_Toc454442807)

[(iii) RESEARCH THE PROBLEM 6](#_Toc454442808)

[(iv) SPECIFIY THE PROPOSED SOLUTION 9](#_Toc454442809)

[(2) Design OF THE SOLUTION [15 marks] 11](#_Toc454442810)

[(i) DECOMPOSE THE PROBLEM 11](#_Toc454442811)

[(ii) DESCRIBE THE SOLUTION 12](#_Toc454442812)

[(iii) DESCRIBE THE APPROACH TO TESTING 13](#_Toc454442813)

[(3) DEVELOPING THE SOLUTION (25 mARKS) 14](#_Toc454442814)

[(i) INTERATIVE DEVELOPMENT PROCESS 14](#_Toc454442815)

[(ii) TESTING TO INFORM DEVELOPMENT 15](#_Toc454442816)

[(4) EVALUATION (20 mARKS) 16](#_Toc454442817)

[(I) TESTING TO INFORM EVALUATION 16](#_Toc454442818)

[(II) SUCCESS OF THE SOLUTION 17](#_Toc454442819)

[(III) DESCRIBE THE FINAL PRODUCT 18](#_Toc454442820)

[(IV) MAINTENANCE AND DEVELOPMENT 19](#_Toc454442821)

[APPENDIX A - BIBLIOGRAPHY 20](#_Toc454442822)

# (1) Anlysis of the problem (10 Marks)

## (i) problem definition

*(a) Describe and justify the features that make the problem solvable by computational methods.*

*(b) Explain why the problem is amenable to a computational approach.*

Invented by German engineer Arthur Scherbius in 1918, the Enigma machine is cipher based device designed to encrypt messages, unable to be read by anyone but the intended recipient with a symmetric key to the message. It was developed and used extensively by the Nazi Government throughout the 30’s, and going into WW2 was one of their most powerful weapons. It allowed German soldiers the capability to send messages over the new radio and morse code quickly and, most importantly, securely. This is why solving the Enigma cipher became a top priority for the Allies, culminating in Alan Turing’s work at Bletchley park, where he invented the Bombe attack in 1940 and consequently greatly advanced the fields of cryptography and computational science. 80 years later, I want to create an educational program to first explain and simulate an Enigma machine so that it can be understood why the machine was so difficult to solve, and in the same program it should be able to solve an Enigma cipher given only ciphertext.

The original Enigma ciphers contained no plugboards, so we shall start here. The process of encrypting a message begins by a button being pressed on the Enigma machines keyboard. This rotates the rotors of the machine according to a pattern, with the right rotor moving every key press, the middle every 26th key press (with double steps) and the leftmost rotor stepping every 26th movement of the middle rotor (or 25th permutation due to a quirk of the enigma machines, double stepping). Electrical current is then passed through the key into the rightmost rotor, which consists of a set of wires essentially substituting the letters. This passes into the next rotor and is repeated until hitting the end, where the electrical current hits a reflector and is substituted yet again, and sent backwards through the rotors (this fact leads to no letter encrypting to itself, a very useful property for code breakers). In later, more complex machines, a plugboard was introduced which would swap two letters at both the input and output, increasing the settings space considerably. Later machines would even have a 4th stationary rotor, and increased variety of rotors. Overall, Bletchley park would need to know: which rotors are in use and at which position at the start of encryption, what offset the rotors had from their stepping position, and the plugboard settings.

Given this, simulating an Enigma machine is a task very well suited to computational methods, given the complicated and involved process that goes into encrypting a letter. A typical encryption requires keeping track of 10 letters, and going through 9 lookups, all while advancing the state of the machine and checking for rotor position changes, and while on paper it can be managed for a few letters, this becomes out of hand very quickly. This is why the original cipher made use of the relatively modern electromechanical system, and a computer can handle this series of operations just as well, being able to store the state of a system very easily and being able to look up substitutions very quickly through array-like structures. Furthermore, an electromechanical system such as was originally used would not be well suited to the task of education due to one simple reason: electricity isn’t easy to see. It becomes very unfeasible very fast to show the internal state of the enigma machine during an encryption due to the sheer number of lights that would be needed – and where they would need to be placed. This is easily solved with a computer, being able to lay out and display the state of the machine without regard to physical dimensions.

Finally, solving an Enigma cipher is incredibly well suited to computational approaches. Due to the straightforward process of ciphering a string, computers can calculate millions of potential keys through iteration, all in a relatively short period. This is far faster than working on paper, with the most effective non-electronic methods for solving Enigma ciphers involving complex plastic sheets and light panels that scaled completely ineffectively, only being practical during the 1930’s when Enigma machines were still only using 3 rotors. Even then, Turing’s Bombe attack was limited in scope and speed, requiring complicated mathematical menus be constructed for each decryption and multiple false starts were produced on the bombe machines per run. In the end, the Bombe attack took somewhere between half an hour to 2 hours per key to decrypt, severely limiting the allies’ responsiveness. In comparison, this technique on modern computers may take somewhere between a minute to a few seconds, orders of magnitude faster.

## (ii) Stakeholders

1. *Identify and describe those who will have an interest in the solution explaining how the solution is appropriate to their needs (this may be named individuals, groups or persona that describes the target end user).*

My program will be designed to appeal to anyone interested in the history of cryptography, but more specifically I will target 16-25 year-olds with a background in maths, linguistics or computer science.

I have 3 specific stakeholders in mind:

Ben Carter – 18 years old, mathematics and computer science student, with backgrounds in cybersecurity. Ben is relevant to this project as they have interests in the history of computers and cybersecurity, wanting to understand the relevance of Enigma to this subject through an understanding of the underlying algorithm and machine.

Sam McWhirter – 18 years old, computer science student, with interest in WW2 history and related Enigma relevance. Sam is relevant to this project as he has had an interest in how the Enigma machine turned the tide of WW2, wishing to see why it was such a difficult problem to decrypt and recognizing the value that the team at Bletchley Park added to the allies in their solving of it.

Henry Warburton – 17 years old, maths and linguistics student, interested in cryptography and has made many hobbyist ciphers and cryptographic systems for fun. He is relevant to this project as he has lots of experience with complex and interesting cryptography throughout history and wants to add yet another cipher to his library of knowledge. However, the currently established websites for this purpose are just not interactive enough for him, requiring following along with long ciphers by paper and pen, not allowing quick feedback when he makes mistakes.

I then asked each of them a questionnaire.

|  |  |  |  |
| --- | --- | --- | --- |
| Question | Ben | Sam | Henry |
| How well do you already understand the engima cipher | I know of the individual components of the machine, but I’m not entirely sure of how the cipher and whole process works | Passing knowledge and understanding. | I understand that it was used in ww2 by the Germans to send secret messages and it was eventually cracked at Bletchley park, however I don’t know how it works |
| Do you understand many Cryptographic ciphers | Yes – simple ciphers such as substitution, one-time pad as well as modern computer ciphers e.g RSA, AES DES | Yes. I understand some simple ciphers, with my most complex being Vigenere cipher | I understand a few rudimentary encoding methods like pig pen and rail fence and caesar ciphers and code wheels and grilles and dot code |
| Could you perform any ciphers on paper | In theory – yes, but modern computer ciphers could take days | Yes, the Caesar cipher. | Probably all of the ones I just mentioned however they’re not very secure so I could also crack them on paper |
| How did you learn about any of these ciphers | Online tools such as cyberchef, a cybersecurity programme called cyberdiscovery | I know this because of “Gravity Falls”, where it was integral part of the lore. | My favourite book when I was younger called the Knowhow Omnibus and it had a section on spying which had lots of information about codes and ciphers |
| Why would you like to learn about the enigma cipher | I enjoy looking at how modern computers can solve a range of old problems, and I am impressed by how these problems were solved in their time. | I love WW2, and I want to learn more about the technology used to destroy the Nazis. | I like cryptology and it seems interesting |
| Are there any features you would like in an enigma cipher educational tool | I’d like to see how data travels through the Enigma. I’d also like to see an estimation of how long it would take for a bombe machine to solve a given cipher. | Text-to-Speech; I am dyslexic | I would like it to show how the physical machine worked and the history of what went into solving how to break it |

As can be seen, the questionnaire shows that despite these people all coming from quite cryptographic/ computer science based areas, none of them have a particularly extensive knowledge of the Enigma machine. Furthermore, they would all be interested in learning them, given the lack of tools online for learning a lot of cryptography, resorting to more technical programs or books to understand the programs, a less refined method of teaching. Finally, I will be sure to include some history of the Bombe attack in my program someway, and some accessibility capabilities should be introduced to the program.

## (iii) RESEARCH THE PROBLEM

*(a) Research the problem and solutions to similar problems to identify and justify suitable approaches to a solution.*

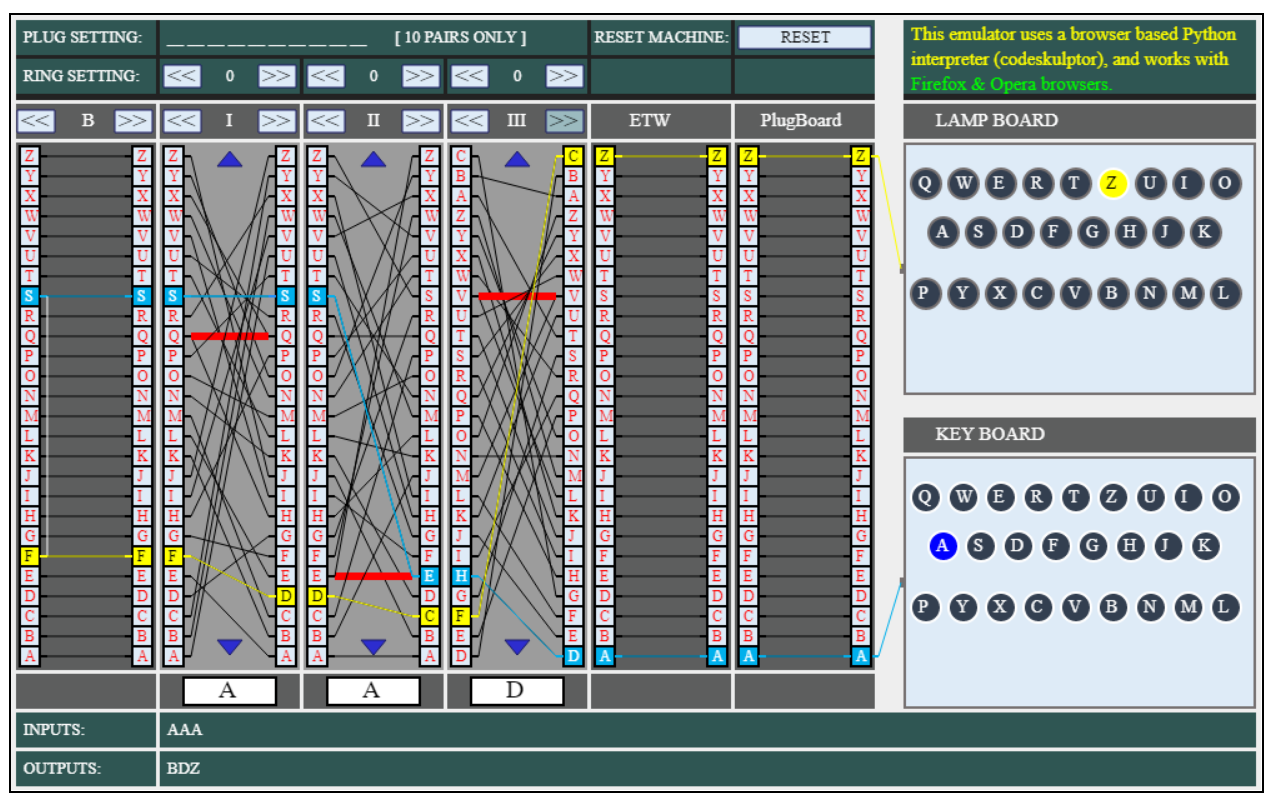
*(b) Describe the essential features of a computational solution explaining these choices.*

*(c) Explain the limitations of the proposed solution.*

<https://piotte13.github.io/enigma-cipher/>

Graphical user interface, diagram

Description automatically generated



*The website encrypting AAA with rotors I II III in position AAA and ring settings 111. As you can see the rotors have mutated and the website uses offset ring settings.*

This website simulates an Enigma machine in your browser. Made in python and interpreted, this site shows the internal workings of the Enigma machine and all it’s functional parts in a visual way. The user can use any of the buttons provided to change a setting such as move/change a rotor or press a key, and can click on 2 plugboard buttons to create plug connection between them. When a key is pressed, that input on the plugboard will light up in blue and each subsequent line through the encryption will light up, changing colour at the reflector to yellow. Each rotor connection is displayed, and rotate in real time in line with the settings provided.  
What I liked, and will try to implement:

* The abstract layout and wire representations
* The colour scheme, with the less vibrant colours being easy on the eyes and professional
* The ability to step through the process and understand what is going on through lit up wires
* The settings of the machine can be manually changed at any point, allowing the user to learn much faster through interaction
* Is an easy to access and use web application, requiring no interpreter or downloads. However I am uncomfortable in my knowledge of web development and don’t think a website can do the heavy lifting required for solving Enigma ciphers, so I am limited to a desktop app

What I disliked about the implementation and will try to change:

* Some settings do not line up with the well-recognized format for enigma settings, eg ring settings are 0 indexed and not 1 indexed like most implementations
* The input system for the settings, while interactive, is quite slow, and I should like the ring/rotor position settings to stay when changing rotor
* Lack of variety of Enigma machine types (only simulating an M3 model)
* Entering a plugboard configuration is very slow and could be easily sped up with a text box
* No solver feature

[https://web.archive.org/web/20060720040135/http://members.fortunecity.com/jpeschel/gillog1.htm](https://web.archive.org/web/20060720040135/http:/members.fortunecity.com/jpeschel/gillog1.htm)

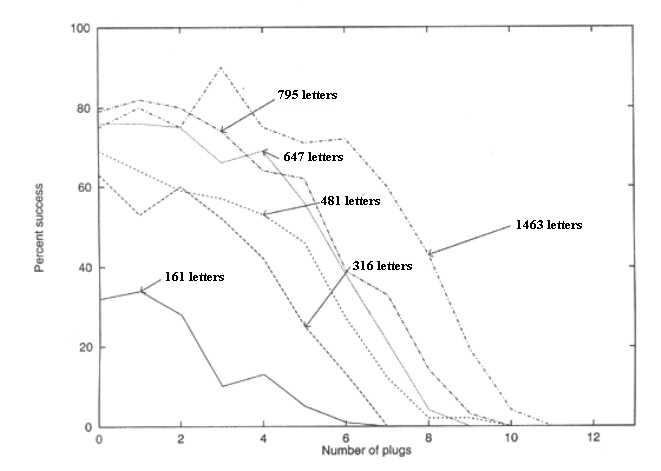
Text

Description automatically generated

A picture containing text

Description automatically generated

*Index of co-incidence formula as given by paper*



*Diagram from paper showing the effectiveness of the algorithm based on length of input and number of plugs. Effectiveness calculated as percentage of inputs deciphered to 80% similarity with plaintext*

This paper from 1995 details an algorithm for solving the Enigma cipher with modern computational techniques, and tests this algorithm on various WW2 ciphers.

What I liked and will implement:

* The stages of the algorithm used, ie solve rotors, solve rings, solve plugboard
* Testing on a wide range of inputs and settings to benchmark the solver
* The fitness functions used, especially earlier on i.e the Index of Coincidence test
* Being able to see the stages of decryption

What I will change in my implementation:

* Have options to repeat certain parts of the algorithm with other known settings, i.e re-solve the rotors after finding the plugboard configuration
* Use a stepped function test for the last stages of the decryption/plugboard hill climb. So instead of using trigrams or bigrams for all 10 plugs, the first 4 could use bigrams, next 4 could use trigrams and the final 2 could use quadgrams, reflecting the effectiveness as more plugs are introduced
* Multithreading could speed up the rotor position finder considerably
* Due to the lack of publicly available historical Enigma messages available, I will not use any for my evaluation and n-gram building like the author of this paper does. Instead, I will use English prose, as that is what will most likely to be given as input.

<https://github.com/mikepound/enigma>

Graphical user interface

Description automatically generated with medium confidence

*A sample deciphering. As can be seen, the correct decryption was the 2nd best rotor configuration. As can be also seen, this was entirely done with the command prompt.*

This is a Java implementation of the previously mentioned paper, made as a demonstration for a computerphile youtube video. The main branch of the code is unoptimized to cater to the youtube audience, but has a more optimized branch making use of multithreading.

What I liked:

* Stages of the decryption process can be seen and understood, with decryption and fitness updates at every stage of the process
* The optimized branch making use of multithreading is very fast, with it only taking a few seconds to fully decrypt a message on my machine – however this may be due to the usage of java, which my design in python may not be able to keep up with
* The code is very modular and object oriented, allowing sections of the algorithm to be modified and mixed and matched very easily
* The usage of numbers instead of letters inside the code allows fast substitutions to be made and math to be performed on the characters without the overhead of switching to and from chars

What I disliked:

* No GUI, or any kind of user interface. All interaction with the program is through java source files
* The algorithm could be improved somewhat, only taking the top result from the rotor selection to the next stage, which statistically is unlikely to be the original rotor settings
* Being written in java, I am unable to use any of the libraries to test my code
* While being very vocal about the stages of the decryption, the program provides somewhat too much feedback during the rotor finding stage, printing to the screen every time a rotor is searched

Limitations

I will develop my program for a python interpreter to run, as it is the language I am most comfortable with and has good support for all areas of the project. However this will lead to some limitations. Namely, I will be unable to produce a high quality, modern GUI and I will be unable to reach particularly fast decryptions, especially on larger Enigma machine variants (e.g the M4) however, to combat this I will write the solver section of the program in C/C++ and call this code to leverage the performance of C code. I may also leverage some web development tools through python libraries to make my interface look a lot better.

Computational solution

Abstraction and visualisation

The key objects will be the machine settings, the machine being emulated and the solver. The rotors can be represented abstractly as lines between letters on either side of the rotor, being moved up and down to represent turning. The plugboard and reflector can represented similarly, but statically.

Thinking ahead

Inputs will be mostly through the mouse and clicking, but some may be done through the keyboard for inputting phrases or settings.

Output will be structured over 3 tabs, one for Enigma emulation, one for specifying the details of the Enigma being emulated and one for solving an Enigma cipher.

Thinking procedurally

The program will be based around the function encrypt(), and will take in a series of settings and a letter and encrypt a letter, while mutating the settings. Multiple encrypt() calls together will construct a messageEncrypt(), and lots of messageEncrypt() with different settings will make up the bulk of the solver algorithm, with stages for first iterating over the rotors (findRotorSettings()), then the rings (findRingSettings()), then the plugboard (findPlugs()). All these problems building on one another makes the problem very amenable to a computational approach, with changing one base procedure propagating up to even the highest level procedures with fixes. Furthermore the messageEncrypt() can be used in the main emulator given that the program stores the state. User interaction will be handled by a various functions being called, but a main updating function will be called in a loop to update the GUI should other functions modify it.

Thinking logically

The critical loop statement for the program will be monitoring for user input on the screen, and updating the interface accordingly. The most critical if-statements will be present in the encrypt() procedure, determining whether to shift the middle and left rotors or, in the internal forms, increase the offset by one.

Thinking concurrently

The main thread will be occupied by updating the screen and handling user input, and can handle encryption so quickly as to seem concurrently, however the solver and file loading may require the main thread create multiple threads to handle the more computationally expensive processes while not impacting performance and responsiveness of the GUI.

## (iv) SPECIFIY THE PROPOSED SOLUTION

1. Aesthetic criteria
   1. The design consists of a default window, with variable width but 600 pixels height and non-resizable by the user
   2. The window should have 3 tabs, a simulator tab, a machine details tab and a solver tab. The tabs should be selectable from a list at top of window
   3. Each tab will follow the same colour scheme of green and white
   4. Text input boxes will follow default windows style, with white background and black text with the effect of seeming indented
   5. Action buttons will follow a similar style, white background and black text but an effect of being raised
   6. Simulator tab
      1. Will have a representation of internal electrical wirings on top left of screen
      2. Colours will consist of green background, with white patches indicating a part of the system and black lines symbolizing electrical wiring. Wires will turn yellow when active
      3. Machine state can be input on right of screen via input text boxes
      4. A string can be put in text box on bottom of screen, and the output is given through another text box below that.
      5. Representation can also be used for input output
      6. Variable number of rotors means a variable width must be given to the tab
      7. Letters will be shown between each stage of the machine so user can follow along
   7. Machine detail tab
      1. Green background, with white boxes for input and output
      2. A box with selectable buttons on left, allowing user to select a machine state
      3. Text box below allows user to give their own filepaths to add to box
      4. Rotors (and reflector) of the machine are display on right with text input, allowing user to add or remove rotors and customize them
   8. Solver tab
      1. Will have a text input box top left for ciphertext
      2. Action buttons below so user can specify which operations to attempt
      3. Text boxes with state below buttons will be filled in in throughout solving
      4. Large simple output terminal showing the state of the solver
2. Input criteria
   1. Tabs can be navigated through and selected through using tab buttons at top
   2. Simulator tab
      1. Buttons above each rotor can rotate through selected rotors
      2. Buttons above and below each rotor allow rotation through rotor positions
      3. Buttons above each rotor can rotate through ring settings
      4. Plugboard selection on right by clicking a letter then clicking another, creating a plug setting
      5. Text boxes for all previously mentioned machine settings
      6. User can use their keyboard to enter a letter and watch it be encrypted
      7. Alternatively, a text box can be used to encrypt a full string
   3. Machine Detail Tab
      1. Saved machine details can be selected by clicking
      2. New machine details files can be added by text box with path to file
      3. Machine details can be saved to file through button
      4. Buttons to add/delete rotors from current details
      5. Reflector will be described by text box
      6. Each rotor described by text box for encryption, smaller text box for stepping positions, and flags to describe which positions the rotor can be in
   4. Solver tab
      1. String input given by large (paragraph size) text box
      2. Grid of buttons allow different operations to be performed on ciphertext
      3. Text box inputs for known machine state inputs for particular operations
3. Output criteria
   1. Each tab will contain a box for outputting feedback in the form of error codes and hints
   2. Simulator tab
      1. A text box row below rotors will output recently encrypted letters and encrypted string
      2. Roman numeral numbers in a box will output the current rotor selection above each rotor
      3. Box with letter above each rotor will display current rotor position
      4. Box with number above each rotor will display current ring position
      5. Each rotor/reflector/plugboard will display all it’s connections, with active ones being lit up (not during full text encryption)
   3. Solver tab
      1. There will be a progress box providing the status of the cracking side
      2. Machine settings will be given by text boxes allowing input and output
      3. The final decrypted message will be given
      4. A box providing console-like output will provide data about each step of the decrypting process, such as fitness scores.
4. Processing criteria
   1. Machine specifications will be loaded from files and validated
   2. All inputs will be validated
      1. Plugboard lengths must be even
      2. Rotor/reflector strings must be 26 letters long
      3. Only 3 or 4 rotors can be given
      4. Rotor selection must be less than the number of available rotors, and greater than 0
      5. Ring/rotor position must be between 1 and 26 inclusive
      6. Only letters can be encrypted – numbers and punctuation ignored
   3. Letters will be encrypted via following steps
      1. Mutate rotor positions, according to stepping positions and double stepping
      2. Encrypt through plugboard
      3. Encrypt through rotors, reflector, rotors
      4. Encrypt back through plugboard
   4. Solver will proceed as follows
      1. Check every combination of rotors and rotor positions, find highest 3 scoring according to IoC fitness function
      2. Check every combination of 2nd and 3rd ring positions on the 3 combinations, keeping top 2 by IoC fitness function
      3. Use a hill climbing technique on both remaining rotor/ring positions to find the optimum plugboard configuration. Output highest scoring settings layout according to fitness functions below
      4. Hill climb according to bi-grams from 1-4 plugs, tri-grams from 5-8 plugs, quad-grams from 9-10 plugs.
   5. Solver must “succeed” 50% of the time with an input of length 500, and complete within 20s with 5 rotors available, 1 minute on 8 rotors available, 2 minutes on 8 rotors available with 4 rotor machine. Tested on my 6-core 3.2ghz pc.
5. Platform
   1. The program must run on a windows 10 64-bit computer, as these are the computers I have access to
   2. The program will require python 3.11+
   3. A mouse will be necessary, and a keyboard recommended
   4. //Here I may need to include the requirments of pybind11, such as cmake, visual studio and the lib itself. I don’t know if this is necessary yet.

# ``(2) Design OF THE SOLUTION [15 marks]

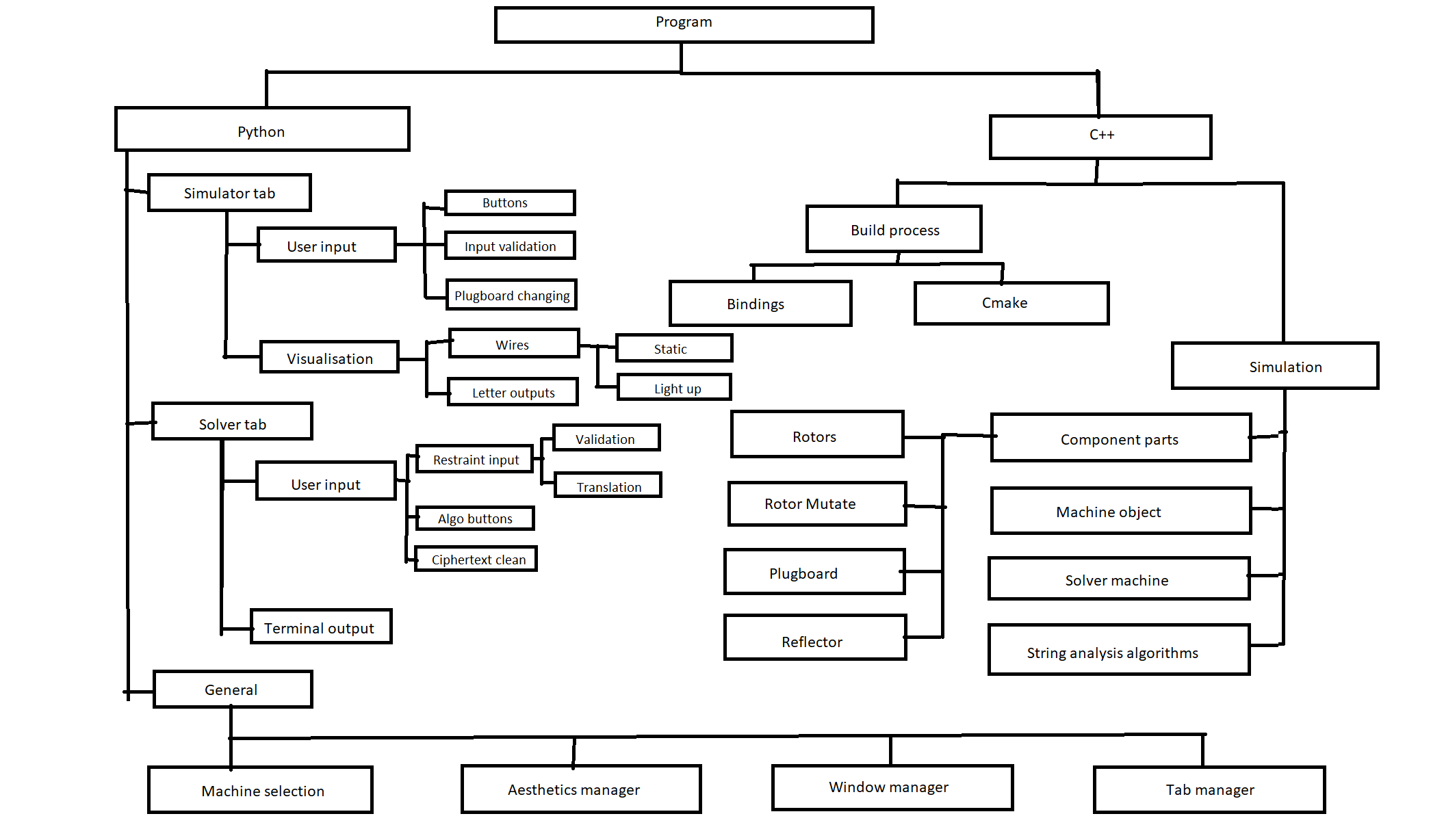
## (i) DECOMPOSE THE PROBLEM

*(a) Break down the problem into smaller parts suitable for computational solutions justifying any decisions made.*

I have chosen to use a mixture of languages. Python will be used for the bulk of the program, allowing me to leverage my strong understanding and fast development times in python to build the GUI and most data handling. However, a large portion of the program will be coded in C++, using a library called “Pybind11” to generate python bindings. This allows me to write performant code in C++, compile it into a DLL and run in the main python program, satisfying the need for very fast code for the brute forcing steps of solving. C++ and python will also be useful for the machine simulation given their hybrid OOP-procedural style.

However, a big separation from the specification must be made here, in the form of abandoning custom machine specifications. This is to ensure the project can be completed in time, as allowing custom specifications outside of the main I, M3 and M4 will over complicate the design, requiring edge cases and customization to be handled in critical areas.

Choose and justify your choice of language – it is NOT good enough to say I will use Python cause school have it and I can use it. That can be mentioned, but you should say – I will use python because it has libraries that I can utilise, such as….I am not too concerned about the GUI element of my solution…..The file handling in Python is relatively easy…..I can adapt Python to become an OOP and this will help me because – this class will contain this object and this object and this object, reducing the amount of code that I need. I will use python in a procedural manner because my project will achieve this task and then this task and so on which lends itself to the procedural nature of this language.



Here is a top-down view of all the modules and parts for this system to work. On the right can be found the functional C++ components and on the left is the python user interface components. This separation allows abstraction, as when creating the python interface all need be done is call a function on an object and the message will be encrypted without having to consider the consequences and process underlying this.

Diagram

Description automatically generated

This is the inheritance/bottom up diagram of how the machine parts will be simulated and ran. Unincluded is the analysis section, coming under float(\*fitness)(int[]), which was originally a function pointer, however during development this will become a class in its own right designated for evaluating possible solutions.

## (ii) DESCRIBE THE SOLUTION

*(a) Explain and justify the structure of the solution.*

*(b) Describe the parts of the solution using algorithms justifying how these algorithms form a complete solution to the problem.*

*(c) Describe usability features to be included in the solution.*

*(d) Identify key variables / data structures / classes justifying choices and any necessary validation.*

*(e) Identified and justified the test data to be used during the iterative development of the solution.*

This is a mockup of the user interfaces. On the left can be seen the internal wirings of the machine, and on the right a panel is used to input data which can be represented on the left.

While creating the mockup, I realized that the specified 600 pixel height would be too small, instead opting to let the user resize the window.

Graphical user interface, chart

Description automatically generated

Graphical user interface, application

Description automatically generated

### Simulation

Simulation exists in C++ and is the functional core of the program. Using an entirely OOP implementation, the key class will be called “Machine” and will contain references to component objects such as Rotors, plugboards and reflectors. Machine will be able to use these objects to abstract away the entire process of rotating rotors, encrypting, converting from int to char and back and a wide range of processes. Throughout the project, any code that requires performance will use integers in place of chars, with 1 = A, 2 = B etc. This prevents a lot of wasteful additions and subtractions being used throughout the code, and less time spent casting to integer types.

#### Wiring

A widely used core class will be “Wiring”. This class’s function is simple. It is initialized by an array of wirings (in integer format) and produces an interface to simple substitute through it. Given that simple substitutions make up most of the operation of an Enigma machine, this will be an important class. It will consist of 2 methods, transform and transform reverse. Transform simply looks up through the wiring field and returns the written int, while transform reverse will create a new, subservient Wiring object with reverse wirings to the original, before calling this Wiring’s transform. This serves to cache a reversed version of wiring, as this will be a result regularly used but still takes quite a bit of computation time to calculate (especially on large scales) .

Class Wiring {

Private int[26] wiring

Private Wiring\* reversePointer = null

Public func Wiring(int[26] wiringIn) wiring = wiringIn endfunc

Public func int transform(int start)

Return wiring[start]

Endfunc

Public func int transformReverse(int start)

If (reversePointer == null)

reversePointer = new Wiring(reverse(wiring))

endif

return reversePointer.transform(start)

endfunc

endclass

#### Components

There will be 3 key derivatives of the Wiring class, Reflector, Plugboard, and Rotor (or Rotor->RotorSpecification)

The reflector will be essentially the exact same as the wiring class, and is unimportant. The plugboard will function very similarly, however it will be constructed differently. Given that the plugboard consists of a list of swapped letters, it only makes sense to construct it with a variable length list of pairs of letters, instead of a wiring system. Also neccess

Finally is the most difficult, the rotor. To aid in solving later, an intermediary class will be used called RotorSpecification. This will store the wirings of a given rotor, but none of its position or rotor attributes. This allows a new rotor be generated quickly for solving, and allow the user to select between loaded possible rotors given a specification containing all the possible rotors (see more in MachineSpecification). The Rotor class will inherit from this RotorSpecification, also storing the position and ring settings of a physical rotor and allowing encryption through itself.

A key part of the rotor class will be the methods Mutate and Transform. Mutate will take in a Boolean and return a Boolean, and will evaluate based on it’s position

#### MachineSpecification

MachineSpecification will be a class loaded at bootup and will define any simulations or any solvers running throughout the program. It will contains vectors of possible rotors and possible reflectors, as well as store information about the machine name. This allows the code to access and change the rotors used at any time quickly and freely. MachineSpecification will also include a way of accessing files describing an enigma machine in JSON format. This allows me to quickly and freely change a specification by loading the JSON file and modifying the wirings in plaintext format (which I can get from <https://www.cryptomuseum.com/crypto/enigma/>). To do this I will use the C++ JSON library by nlohmann (found at <https://github.com/nlohmann/json>)

### Solution

### Presentation

Diagram

Description automatically generated

**Pseudo code**

Proc calculator\_main

Open main\_page

If x < 7

Open random\_message(“this must be a binary number”

End if

………………………………….etc

In order to test the functionality of this page, I will test the following items

**Button1**

Should close the main page and open the denary – binary converter

**Button 2**

Should close the main page and open the binary – denary converter

………………………etc

You need to plan the tests that you will use as you are developing this module – it can be quite short – tests should be designed to demonstrate that your system is working – particularly validations and the logic of the module.

**Test Plan for development**

**For var\_number**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Test type | Predicted outcome | Actual outcome | Pass/Fail |
| 010100101 | valid | Saves number |  |  |
| 01010100101001010100 | borderline | Saves number |  |  |
| 000000000000 | boarderline | Saves number |  |  |
| bob | Invalid | Rejects text |  |  |

………..etc

## (iii) DESCRIBE THE APPROACH TO TESTING

1. *Identify the test data to be used during the iterative development and post development phases and justify the choice of this test data.*

The system will be tested in many ways. The first that will be used is to test the functionality of the underlying functional code using a compiled debug executable. This will allow me to adapt and test the code iteratively while working on the functionality, as opposed to having to connect the C++ code to python and a GUI. The functional code will not have any validation, so no invalid or borderline testing will be done here, only testing for functionality, using bigger and bigger tests to cover more and more failure points.

This is the overall testing for the system and should be a test plan that proves that you have completed the system and met all of your success criteria. You need to describe your methods of testing – these could include alpha, beta, white box, black box, top down & bottom up. There are several ways in this should be achieved:

1. System test – derive three scenarios that need to be completed, e.g. a customer needs their details entered, they wish to make a purchase and get their receipt. Choose one to be valid, the next to be borderline and the last to have invalid data in it.
2. Ask you user to test your system. Give them two tasks to do so that they can experience the usability – acceptance testing (black box)
3. White box testing – stress test variables & functions to try and expose weaknesses, e.g.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Test type | Predicted outcome | Actual outcome | Pass/Fail |
| 010100101 | valid | Saves number |  |  |
| 01010100101001010100 | borderline | Saves number |  |  |
| 000000000000 | borderline | Saves number |  |  |
| bob | Invalid | Rejects text |  |  |

# (3) DEVELOPING THE SOLUTION (25 mARKS)

## (i) INTERATIVE DEVELOPMENT PROCESS

*(a) Provide annotated evidence of each stage of the iterative development process justifying any decision made.*

*(b) Provide annotated evidence of prototype solutions justifying any decision made*.

## (ii) TESTING TO INFORM DEVELOPMENT

*(a) Provide annotated evidence for testing at each stage justifying the reason for the test.*

*(b) Provide annotated evidence of any remedial actions taken justifying the decision made.*

# (4) EVALUATION (20 mARKS)

## (I) TESTING TO INFORM EVALUATION

*(a) Provide annotated evidence of testing the solution of robustness at the end of the development process.*

*(b) Provide annotated evidence of usability testing (user feedback).*

## (II) SUCCESS OF THE SOLUTION

*(a) Use the test evidence from the development and post development process to evaluate the solution against the success criteria from the analysis.*

## (III) DESCRIBE THE FINAL PRODUCT

*(a) Provide annotated evidence of the usability features from the design, commenting on their effectiveness.*

## (IV) MAINTENANCE AND DEVELOPMENT

(*a) Discuss the maintainability of the solution.*

*(b) Discuss potential further development of the solution.*

## APPENDIX A - BIBLIOGRAPHY