Electronics and
Computer Science

UNIVERSITY OF
Southampton

# Coursework Specification

| | | | |
|---|---|---|---|
| **Module:** | COMP1202: Programming 1 | | |
| **Assignment:** | Programming coursework | **Weighting:** | 40% |
| **Lecturers:** | David Millard (dem), Mark Weal (mjw) | | |
| **Deadline:** | 5/Dec/2017 | **Feedback:** | 12/Jan/2018 |

## Coursework Aims

This coursework allows you to demonstrate that you:

- Understand how to construct simple classes and implement methods.
- Are able to take simple pseudo-code descriptions and construct working Java code from them.
- Can write a working program to perform a complex task.
- Have an understanding of object oriented programming.
- Can correctly use polymorphism and I/O.
- Can write code that it is understandable and conforms to good coding practice.

## Contacts

General programming queries related to the coursework should be made to your demonstrator in the timetabled labs.

Queries about the coursework specification should be made to Mark Weal (mjw@ecs.soton.ac.uk).

Any issues that may affect your ability to complete the coursework should be made known to Mark Weal (mjw@ecs.soton.ac.uk) or David Millard (dem@ecs.soton.ac.uk), ideally before the submission deadline.

## Instructions

Late submissions will be penalised at 10% per working day.
No work can be accepted after feedback has been given.
You should expect to spend up to 50 hours on this assignment.
Please note the University regulations regarding academic integrity.

# ECS Enigma machine simulator

## Specification

The aim of this coursework is to construct a virtual representation of the Enigma machine, an electro-mechanical rotor cipher machine used for the encryption and decryption of secret messages. In constructing the machine you will be simulating the key physical components of the machine (Rotors, Reflectors, Plugs and Plugboard) enabling you to encode and decode text messages. Figure 1 below shows an example Enigma machine.

For this coursework you are expected to follow the specification of the machine as set out below. Where class names, variables or methods are highlighted in red, we expect the naming to follow the specification exactly. This specification may not correspond exactly to the enigma machine in reality or indeed optimal ways of modelling the machine, but we have chosen aspects of the machine to model that help you to demonstrate aspects of Java Programming.

**Your task is to implement the specification as written.**

**Please do NOT use java packages as this complicates the marking process.**

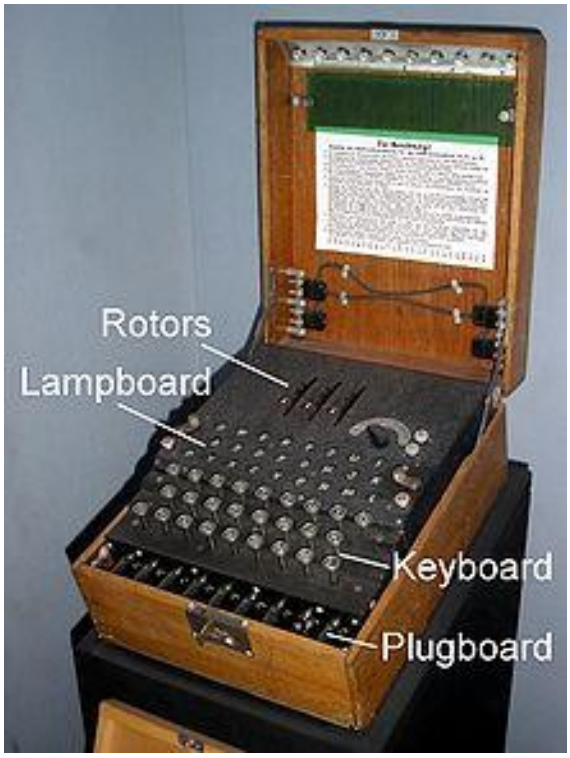


Fig 1. The Enigma machine (Karsten Sperling, 2004)

## How the Machine Works

The operator of an Enigma machine presses one of the keys on the keyboard (A-Z) and this results in one of the bulbs lighting up (A-Z). The lit bulb represents the encoded character. Which character a key encodes to is dependent on the electrical path formed through the various components of the machine. A simplified representation of this is shown below in Figure 2. An example trace is shown through the components of

the machine illustrating how the letter to be encoded is mapped by the different components as it passes through the mechanisms of the machine.

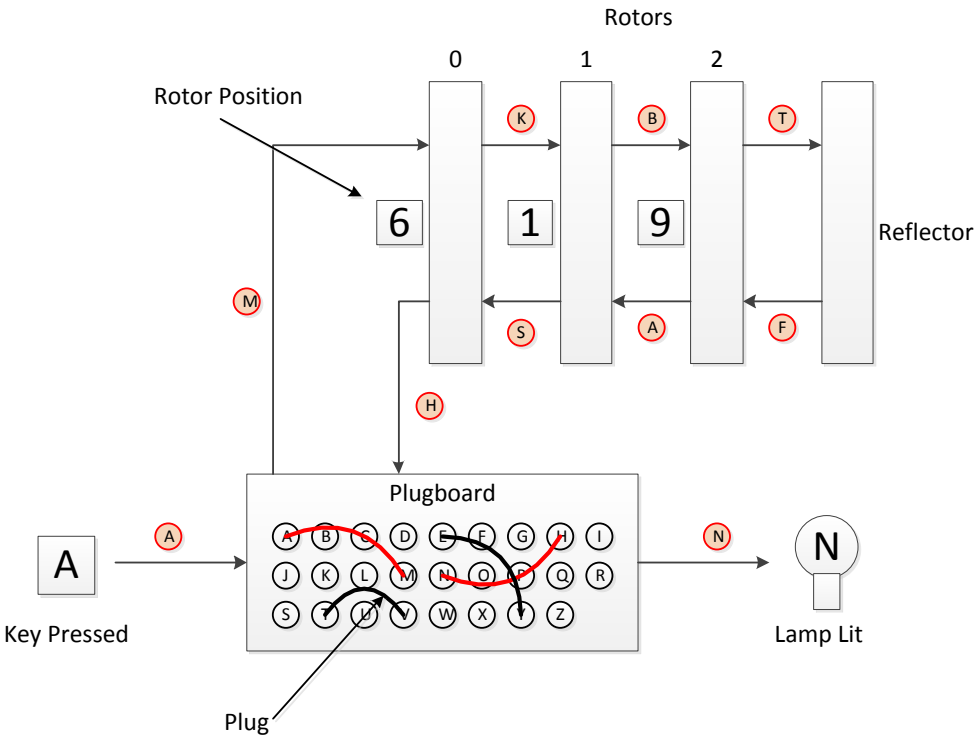


Fig 2. A simplified diagrammatic representation of the machine.

The reason that the Enigma machine was such an effective encoding machine was down to the components from which it was constructed. These include:

**The Plugboard:** Contains 26 sockets, one for each letter of the alphabet. Plugs could be inserted into the Plugboard that connected any two sockets. Accordingly, up to 13 plugs could be connected into the plugboard. If a letter is passed to the plugboard and a plug is connected to that letters socket then it is substituted for the letter connected at the other end of the Plug.

**The Rotors:** The basic Enigma machine had three Rotor slots. Each Rotor has 26 pins on one side that are connected to 26 pins on the other side to create a mapping. Any character passed to the Rotor is substituted by the character of the corresponding pin on the other side. Each Rotor can be set to one of twenty six positions at the start of encoding. After each letter has been encoded the first of the three Rotors is rotated by one position changing the mapping accordingly. This ensures that the same letter won't subsequently be encoded the same way. To further add to the complexity some Rotors are **Turnover Rotors**. If after being rotated the first Rotor is at its turnover position then it will cause the Rotor to its right to rotate one position. This was carried out using a ratchet and pawl system. The Turnover Rotors meant that when encoding longer messages all the rotors will change position at some point. There were many different types of Rotor in terms of mappings but we will just model five different Basic Rotors and five Turnover Rotors.

**The Reflector:** The Reflector is placed on the end of the three Rotors and acts like a simple Rotor in mapping its input to a different output. However, the output is directed back to the three Rotors. It is the Reflector that makes the Enigma machine self-reciprocal, i.e. encoding is the same as decoding.

The reason that the Enigma code proved so hard to break was that in order to decode a message you needed to know.

- Which of the different Rotors had been placed in the three slots of the machine.
- What the initial position was of each of these rotors.
- What plug connections were made to the Plugboard.

The following pseudo-code explains the process of encoding a single character. This will later form the basic algorithm of your machine.

*A character (A-Z) is entered by the operator by pressing one of the keys.*
*This character is sent to the Plugboard.*
*If there is a Plug connected to the Plugboard socket corresponding to that letter*
    *The letter is changed to that of the other end of the Plug.*
*Otherwise, the original letter is passed on.*
*The output of the Plugboard is sent to the first Rotor.*
*The mapping of the Rotor changes the letter to a different letter.*
*This letter is passed into the second Rotor.*
*The mapping of the second Rotor changes the letter.*
*The new letter is passed to the third Rotor.*
*The mapping of the third Rotor changes the letter.*
*The third Rotor the letter passes to the Reflector.*
*The Reflector maps the letter and reflects it back to the third Rotor.*
*The third Rotor changes the letter again. This time, an inverse mapping is used as the signal is passing the other way through the Rotor.*
*The output goes to the second Rotor and this changes the letter with its inverse mapping and passes it back to the first Rotor.*
*The first Rotor uses its inverse mapping to change the letter and passes the resulting letter back to the Plugboard.*
*if there is a Plug in the socket corresponding to the letter*
    *The letter is changed to that of the other end of the Plug.*
*Otherwise, the received letter is passed on.*
*The resulting encoded letter is displayed.*

    *[We now need to rotate the rotors]*
*The First Rotor is rotated by one increment.*
*If this Rotor is a TurnoverRotor AND its position corresponds to its turnoverPosition*
    *Then rotate the second Rotor.*
    *If Rotor two is a TurnoverRotor AND its position corresponds to its turnoverPosition*
        *Then rotate the third Rotor.*

Before coding up the encoding algorithm however it will be necessary to construct the basic components of the Enigma machine. The next sections will take you through the construction of the various components.

## Part 1 – Modelling the Plugboard

The plugboard sits at the front of the machine and provides the first encoding mechanism of the machine. The plugboard has 26 sockets, one for each letter of the alphabet. Plugs (leads) can then be used to connect any two sockets together. The original Enigma machines could have up to 13 plugs (no letter can be connected twice!).

You will need to create two classes to model the plugboard, these should be called *Plugboard* and *Plug*.

A *Plug* connects two sockets so you will need to create a constructor that takes two parameters, the end sockets of the plug. You will also want to create some 'get' and 'set' methods to get the ends of the plugs, *getEnd1(), getEnd2(), setEnd1(char), setEnd2(char)*.

The way the encoding and decoding works is that when a letter is keyed in, if there is a plug connected to that letter socket the plugboard returns the value of the letter at the other end of the plug.

Add a *encode(char letterIn)* method to your *Plug* class which returns the *letterIn* if one of the ends doesn't match letterIn, or if one end does match, returns the letter at the other end of the plug. The *Plugboard* can use this to check an individual plug.

The final method you will need for your *Plug* class is called *clashesWith(Plug plugin)*. This method is used to test whether a plug can be connected to the *Plugboard* or whether one of the sockets is already in use. The method should return true if either end of the *Plug* is shared with the *Plug* passed into the method.

You should now have constructed a simple *Plug* class.

Your *Plugboard* class will contain up to 13 *Plugs.* The *addPlug()* method should take two characters representing the ends of the plug and create a *Plug* connecting those characters returning *true*, or, if it clashes with an existing *Plug* it should return *false*.

You should implement some access methods for the *Plugboard*. These should include *getNumPlugs()* which returns the number of plugs currently connected to the *Plugboard* and *clear()* which removes all the plugs from the *Plugboard*.

Your *Plugboard* will also require a *substitute(char)* method that the Enigma machine can use to ask the *Plugboard* to encode a character. If there is an appropriately connected plug then the character returned is encoded, otherwise it returns the character it was passed.

You should now have a *Plugboard* class.

## Part 2 – Modelling the Rotors

To model the various types of rotor in the Enigma machine (*Reflector*, *BasicRotor* and *TurnoverRotor*) you will need to use inheritance.

The following simple diagram shows you how the four classes that you will create are related to each other.

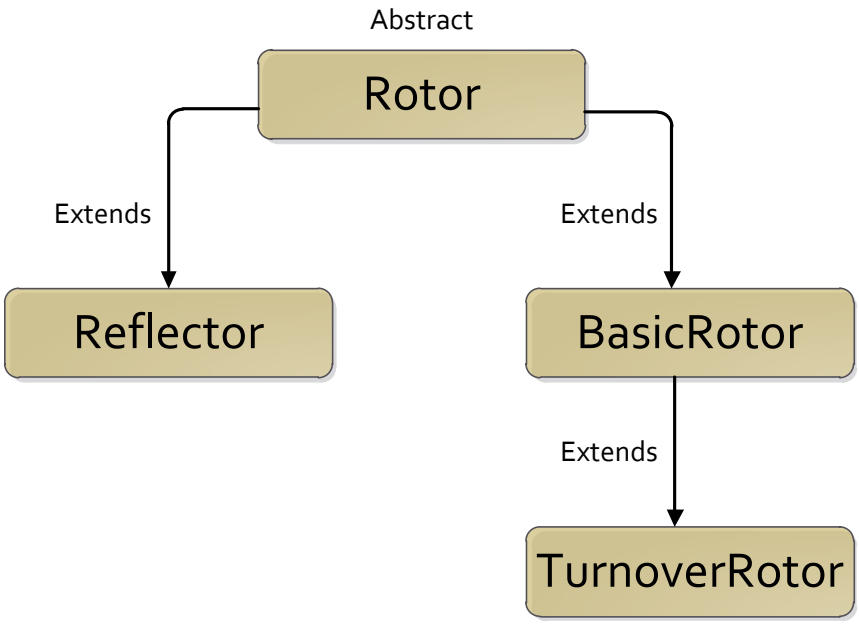


Fig 3: Class diagram for the rotor classes.

The first class to create is the *Rotor* class. This is an abstract class that defines the basic properties and methods that all the other rotor classes will use. The properties that you will need to define are:

- name – the type of rotor as a String ("I","II","III","IV","V","ReflectorI","ReflectorII")
- position – the current position of the rotor, which will have a value between 0 and 25 representing the letters of the alphabet.
- mapping – The mapping of the rotor (an array of integers).
- ROTORSIZE – a constant that defines the number of positions on a rotor (26).

The abstract rotor class will also define two access methods for *setPosition(int)* and *getPosition()*.

Finally, you will need to define two functions that the other rotor classes can override. These functions are *initialise(String)* and *substitute(int)*.

## Part 3 – Modelling the Reflector

The simplest of the Rotors to model is the *Reflector*. The *Reflector* class extends the *Rotor* abstract class. The reflector has two possible types, ReflectorI and ReflectorII, each of which has a different mapping. This mapping is set when the initialise function is called. The mappings are given below as arrays of integers.

ReflectorI

{ 24, 17, 20, 7, 16, 18, 11, 3, 15, 23, 13, 6, 14, 10, 12, 8, 4, 1, 5, 25, 2, 22, 21, 9, 0, 19 }

ReflectorII

{ 5, 21, 15, 9, 8, 0, 14, 24, 4, 3, 17, 25, 23, 22, 6, 2, 19, 10, 20, 16, 18, 1, 13, 12, 7, 11 }

The *substitute(int)* method should return the number corresponding to that element of the array. In order to simplify the processing of the characters as they pass through the system the rotors use the number 0-25 to represent the characters A-Z. This is intended to simplify the internals of the processing.

You should now have a *Reflector* class.

## Part 4 – Modelling Basic Rotors

You now need to model a *BasicRotor* class. This class will also inherit from *Rotor*. The basic rotors come in five different types in our machine, represented by roman numerals. The mappings for the rotors are given below.

I = { 4, 10, 12, 5, 11, 6, 3, 16, 21, 25, 13, 19, 14, 22, 24, 7, 23, 20, 18, 15, 0, 8, 1, 17, 2, 9 }

II = { 0, 9, 3, 10, 18, 8, 17, 20, 23, 1, 11, 7, 22, 19, 12, 2, 16, 6, 25, 13, 15, 24, 5, 21, 14, 4 }

III = { 1, 3, 5, 7, 9, 11, 2, 15, 17, 19, 23, 21, 25, 13, 24, 4, 8, 22, 6, 0, 10, 12, 20, 18, 16, 14 }

IV = {4, 18, 14, 21, 15, 25, 9, 0, 24, 16, 20, 8, 17, 7, 23, 11, 13, 5, 19, 6, 10, 3, 2, 12, 22, 1 }

V = { 21, 25, 1, 17, 6, 8, 19, 24, 20, 15, 18, 3, 13, 7, 11, 23, 0, 22, 12, 9, 16, 14, 5, 4, 2, 10 }

Your *BasicRotor* class will need a constructor that takes the type as a string.

You will need to provide an implementation for the *substitute(int)* method that takes an integer representing a letter and returns an integer represented by that position in the mapping. You will have to take into account the position of the rotor when making this substitution however. If the rotor is at position

0 then the mapping is used as is. If the rotor is at position 10 then you will need to remove 10 from the character being mapped before carrying out the mapping and add 10 back on after the mapping.

You will also need to implement *a substituteBack(int)* method that uses an inverse mapping to change the integer passed to it. You can create the *inverseMapping* as an array from the *mapping* array. If *mapping*[x] = y, then *inverseMapping*[y] = x.

The final method to implement for *BasicRotor* is *rotate()*. This advances the position of the rotor by one. Again, remember that these rotors are circular. If the position is advanced to the ROTORSIZE then you need to set it back to 0 as the rotor has performed a complete revolution.

You should now have implemented a *BasicRotor*.

## Part 5 – Modelling the Enigma Machine

You are now ready to create an *EnigmaMachine* class. The machine is composed of a *Plugboard*, Three *BasicRotor*s in positions 0, 1 and 2, and a *Reflector*.

You will need to create a constructor for the *EnigmaMachine*, this should create an empty *Plugboard*.

You will now need to create the following methods.

*addPlug(char,char)*

*clearPlugboard()*

*addRotor(BasicRotor,slot)*

*getRotor(slot)*

*addReflector(Reflector)*

*getReflector()*

*setPosition(slot,position)*

The final method you will need to create is *encodeLetter(char)*. The pseudo-code presented earlier gives you the algorithm to use inside *encodeLetter*. You can pass a char into the plugboard, but will need to convert the output to an integer representation (0-25) for use with the rotors and reflector. Once you have received the output from the first rotor after it being passed back you will need to convert It back to a character before the final pass back through the plugboard. The enocdeLetter method should only rotate the first rotor (0).

You should now be able to create a start() method with which to test your *EnigmaMachine* class. The following example messages and settings can all be decoded to reveal the names of animals.

**Test 1**

Plugs [A-M] [G-L] [E-T]

Rotors slot 0 = BasicRotor typeI , initial position 6

Slot 1 = BasicRotor typeII , initial position 12

slot 2 = BasicRotor typeIII , initial position 5

ReflectorI

Encoded message = GFWIQH

**Test 2**

Plugs [B-C] [R-I] [S-M][A-F]

Rotors slot 0 = BasicRotor typeIV , initial position 23

Slot 1 = BasicRotor typeV , initial position 4

slot 2 = BasicRotor typeII , initial position 9

ReflectorII

Encoded message = GACIG

If you have correctly identified the two animals in test 1 and 2 then congratulations, you now have a working enigma machine with basic rotors.

## Part 6 – Reading and Writing Files

You now have an Enigma machine for encoding and decoding characters but you may wish to encrypt and decrypt longer messages. To make this easier you should create a new class called *EnigmaFile* that creates an EnigmaMachine , reads in a file containing text, encrypts this file using the enigma machine and writes the output back to a different file.

## Part 7 - Modelling Turnover Rotors

Having now modelled the *BasicRotor* class, this can be extended to create the *TurnoverRotor* class. The Turnover rotor has a *turnoverPosition* that is defined for each type of rotor. These turnover positions are as follows for our version of the machine.

typeI=24, typeII=12, typeIII=3, typeIV=17, typeV=7

When *rotate()* is called on our turnoverRotor it will rotate one position in the same way as the BasicRotor. However, if its new position is the same as the turnoverPosition, it will then call rotate() on the Rotor to its right in the machine. To do this, you will need to add a nextRotor property to the class that will be used to store the next rotor if the rotor is placed in slot 0 or 1 (the Reflector is not rotated). You may wish to add a new method called *setNextRotor()* and/or add a new constructor which takes a Rotor in addition to the type. The nextRotor property is not needed for basicRotors so should only be implemented in the extended class.

The mappings of the turnover rotors are the same as those for the basic rotors of the same type so these should be inherited from the BasicRotor.

Once you have implemented the TurnoverRotor class you should be able to run the following test to make sure your turnover rotors are working properly. Notice, that you need a longer message in order to ensure that at least two of the rotors turn. Again, the output should be about animals and their related activites.

**Test 3**

Plugs [Q-F]

Rotors slot 0 = TurnoverRotor typeI , initial position 23

Slot 1 = TurnoverRotor typeII , initial position 11

slot 2 = turnoverRotor typeIII , initial position 7

ReflectorI

Encoded message = OJVAYFGUOFIVOTAYRNIWJYQWMXUEJGXYGIFT

## Part 8 – Building a Bombe

You now have an enigma machine simulator, and are in a position to encode and decode messages. However, what made the difference at Bletchley Park, and led in part to the development of modern computing as we know it was the machine they built to help decode messages when they didn't know the settings on the Enigma machine. This was known as the Bombe (https://en.wikipedia.org/wiki/Bombe).

For the last part of your coursework, the aim is to build a simple test harness for your Enigma machine that can run through different permutations of plugs and rotors in order to try and decrypt messages. To do this, you will need to write a Java class that creates an enigma machine, then repeatedly tries different settings on it with an encoded message until it outputs the decrypted version of the message. To get you going, there are three challenges below that each focus on just testing different aspects of the settings. To help you know when you have successfully decoded the message we have given you one of the words contained in the decrypted message. When you find this word present in the string you output you may have the correctly decoded message. It is fine to create quite bespoke Bombes, don't be too ambitious and try to create very generic solutions.

In each case below some of the key setting information has been mislaid. Can you develop a code breaking harness for your machine that will enable you to discover the correct settings. We have included words that we suspect might occur in the original message to help you work out when you may have achieved a solution. If you successfully complete the challenges, document your answers in the readme file include in your coursework submission.

**Challenge 1 : Some of the plug ends fell out!**

Plugs [D-?] [S-?]

Rotors slot 0 = BasicRotor typeIV , initial position 8

Slot 1 = BasicRotor typeIII , initial position 4

slot 2 = BasicRotor typeII , initial position 21

ReflectorI

Encoded message = JBZAQVEBRPEVPUOBXFLCPJQSYFJI

May contain the word ANSWER

**Challenge 2 : What were the rotor positions!**

Plugs [H-L] [G-P]

Rotors slot 0 = BasicRotor typeV, initial position ?

Slot 1 = BasicRotor typeIII , initial position ?

slot 2 = BasicRotor typeII , initial position ?

ReflectorI

Encoded message =
AVPBLOGHFRLTFELQEZQINUAXHTJMXDWERTTCHLZTGBFUPORNHZSLGZMJNEINTBSTBPPQFPMLSVKPETWFD

May contain the word ELECTRIC

**Challenge 3 : Which rotors were used!**

Plugs [M-F] [O-I]

Rotors slot 0 = BasicRotor type? , initial position 22

Slot 1 = BasicRotor type? , initial position 24

slot 2 = BasicRotor type? , initial position 23

ReflectorI

Encoded message = WMTIOMNXDKUCQCGLNOIBUYLHSFQSVIWYQCLRAAKZNJBOYWW

May contain the word JAVA

## Extensions

Having constructed your Enigma machine you may wish to add some extensions. You are free to extend your code beyond the basic simulation as described above. You are advised that your extensions should not alter the basic structures and methods described above as marking will be looking to see that you have met the specification as we have described it. If you are in any doubt about the alterations you are making please include your extended version in a separate directory.

Some extensions that you might like to include:

- You could extend your machine to allow a user to type a message a key at a time and have a translation of the key appear in the command line.
- You could add a command line interface you your machine to allow the user to specify the plugs in the machine and the type and initial positions of the rotors.
- You could include the facility to convert the input file into appropriate text for encoding/decoding. This might involve removing any whitespace, capitalising all letters in the file and removing any characters that can't be encoding including punctuation.
- You might wish to extend your Bombe so that it can attempt to uncover messages where it doesn't know in advance what words are contained in the message. To achieve this you might try testing your decoded messages against a dictionary of common words to try and find matches.
- You could use polymorphism to create a subclass of BasicRotor that used a different mechanism to do the mapping
- You could use generics to allow your Engima machine to translate arbitrary types from one to another, not just characters.

15% of the marks are available for implementing a reasonable sized extension. Any of the above examples would be suitable, but feel free to come up with one of your own if you like. It is not necessary to attempt lots of extensions in order to gain these marks but complexity will be considered, so simply adding a new rotor with a different mapping would not in itself be enough to gain all the marks. Please describe your extension clearly in the readme file that you submit with your code along with any special instructions required to run it. If the extension requires deviating from the original spec then please create a new version in a different directory.

## Exceptions

Although it is not a requirement for marks, you could also try to use exceptions in the construction of your simulation. It will be necessary to appropriately catch I/O exceptions in your file handling code, but you also might consider the use of exceptions to correctly handle:

- The input of non-compliant characters (not A-Z).
- Attempts to use a particular rotor type twice (you only have one of each rotor).
- Missing rotors on encoding.
- Attempting to put two plugs into the same socket.
- Missing or inappropriate files.

### Space Cadets

You might want to add a Gui to your Enigma machine to enable the user to carry out the various acts of an Enigma machine operator (adding plugs, choosing rotors, setting rotor positions, encoding individual letters. **No marks are available for a GUI**, we put the suggestion forward simply for the challenge of it. If you attempt this, please save your Gui augmented enigma machine in a separate directory so as not to confuse it with your main submission.

## Submission

Submit **all Java source files** you have written for this coursework in a zip file **enigma.zip**. **Do not submit class files.** As a minimum this zip will include:

Plug.java, Plugboard.java, Rotor.java, BasicRotor.java, TurnoverRotor.java, Reflector.java, EnigmaMachine.java, Bombe.java.

Also include a text file **enigma.txt** that contains a brief listing of the parts of the coursework you have attempted, describes how to run your code including command line parameters, and finally a description of the extensions you have attempted if any. Submit your files by **Tuesday 5th December 2017 16:00** to: https://handin.ecs.soton.ac.uk

## Relevant Learning Outcomes

1. Simple object oriented terminology, including classes, objects, inheritance and methods.
2. Basic programming constructs including sequence, selection and iteration, the use of identifiers, variables and expressions, and a range of data types.
3. Good programming style
4. Analyse a problem in a systematic manner and model in an object oriented approach

## Marking Scheme

| Criterion | Description | Outcomes | Total |
|---|---|---|---|
| Compilation | Applications that compile and run. | 1 | 20 |
| Specification | Meeting the specification properly. | 1,2,4 | 40 |
| Extensions | Completion of one or more extensions. | 1, 2,4 | 15 |
| Style | Good coding style. | 3 | 25 |