Enigma Machine - Documentation

Author, Developer: Zoheb Hasan, Stony Brook University, Computer Science - 114687894
September, 17th, 2024
System Design Link

1. Overview

This simulation of the Enigma Machine follows a modular architecture that mimics the functionality of the original Enigma Encryption device used during World War II. The core components include rotors, plugboards, and reflectors, all contributing to a complex substitution cipher system.

Main Components

- `EnigmaMechine` class.
- `PlugBoard` class.
- `Rotor` class.
- `KeyPair` class.
- `Key` class.
- `Reflector` class.

2. Key Classes & Methods

2.1 EnigmaMachine Class

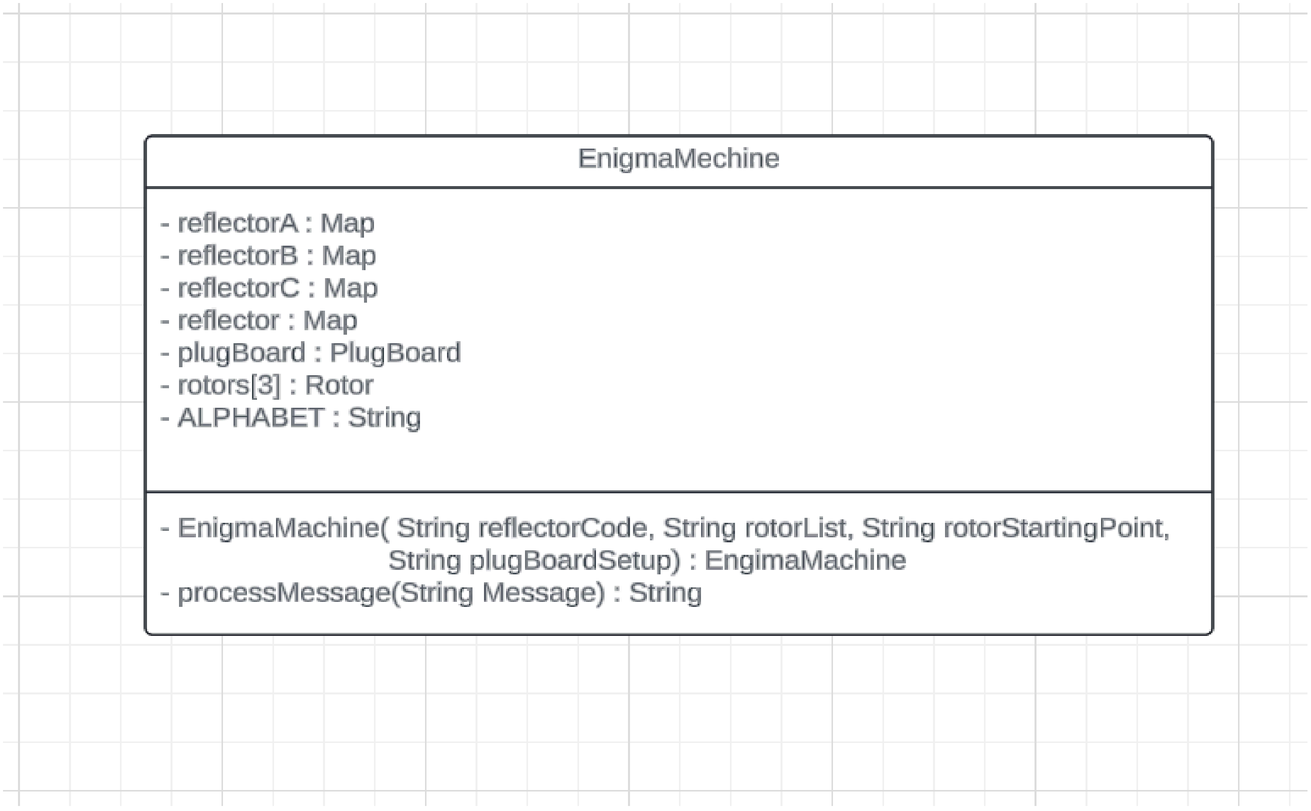
This is the core class responsible for handling the Enigma machine's functionality in my implementation, including initialization, processing input, and managing encryption through rotors and reflectors.

Attributes

- `reflector`
- `plugBoard`
- `staticBoard`
- `rotors`

Key Methods

- `processMessage(String message)`
  `return: String`


EnigmaMechine
- reflectorA : Map
- reflectorB : Map
- reflectorC : Map
- reflector : Map
- plugBoard : PlugBoard
- rotors[3] : Rotor
- ALPHABET : String
- EnigmaMachine(String reflectorCode, String rotorList, String rotorStartingPoint, String plugBoardSetup) : EngimaMachine
- processMessage(String Message) : String

2.2 PlugBoard Class

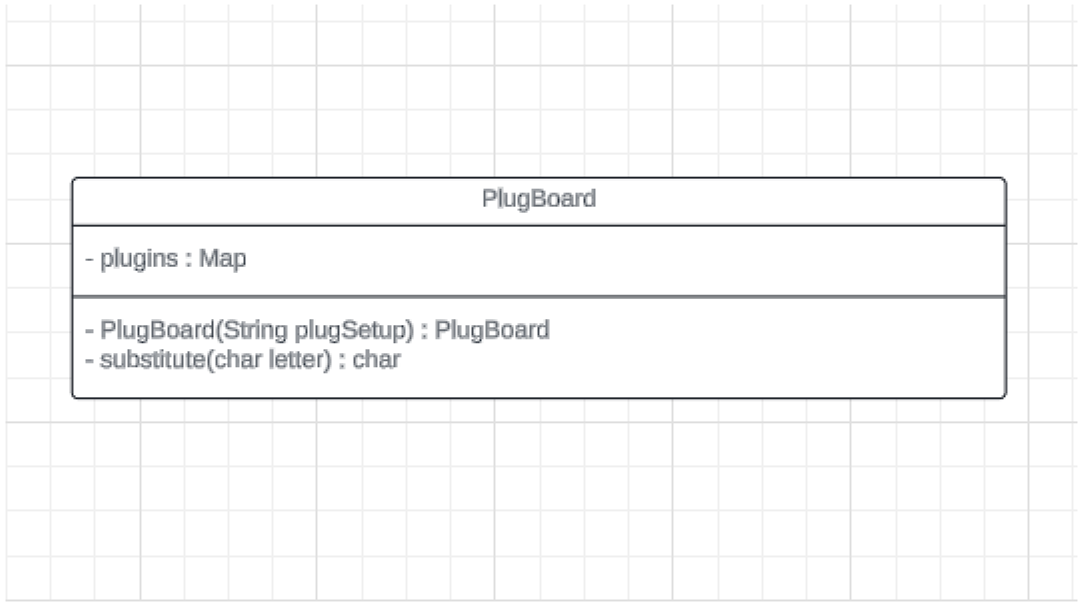
This class simulates the plugboard component of the Enigma machine, allowing users to configure pairwise character swapping.

Attributes

- plugins: `Map<Character, Character>` that holds character pairs for substitution.

Key Methods

- substitute (char letter): Looks up the provided letter in the plugboard map and swaps it if a corresponding pair exists.


PlugBoard
- plugins : Map
- PlugBoard(String plugSetup) : PlugBoard
- substitute(char letter) : char

2.3. Rotor Class

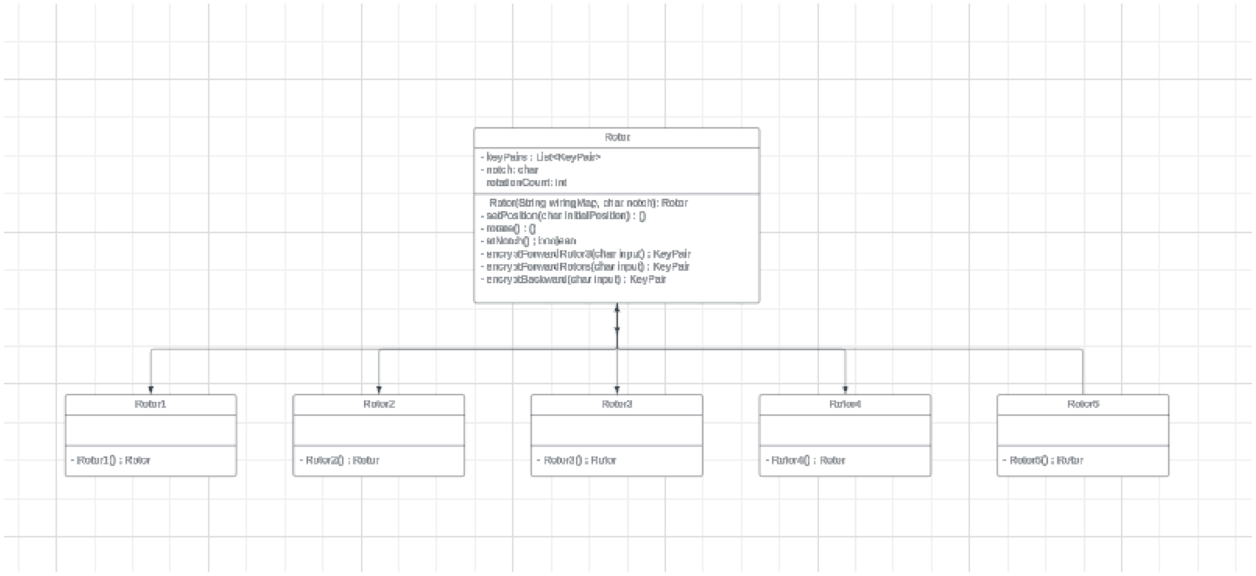
The Rotor class is responsible for forward and backward character substitution through the rotor, which consists of character mapping that changes dynamically as the rotor rotates.

Attributes

- `keyPairs`: A list of `keyPair` objects respecting the character mapping of the rotor.
- `notch`: The position that triggers the next rotor to rotate when reached.

Key Methods

- `setPosition(intialPosition)`: Adjust the rotor's starting position by moving the character with the matching right key to the front
- `rotate()` : Rotates the rotor by one position.
- `encryptForwardRotor3(input, index)`: Handles the forward encryption of a character when it passes through rotor 3.
- `encryptForwardRotors(input, index)`: Handles the forward encryption of a character when it passes through rotors other than `rotors[3]`.
- `encryptBackward(output)`: Handles the forward encryption of a character when it passes from left to right.

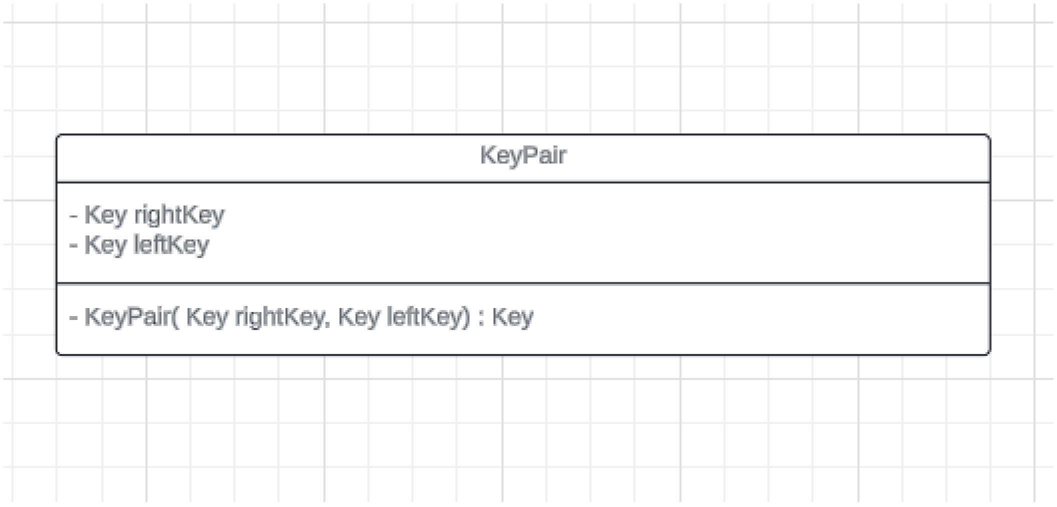

Rotor
- keyPairs : List<KeyPair>
- notch: char
rotationCount: int
Rotor(String wiringMap, char notch): Rotor
- setPosition(char initialPosition) : ()
- rotate() : ()
- atNotch() : boolean
- encryptForwardRotor3(char input) : KeyPair
- encryptForwardRotors(char input) : KeyPair
- encryptBackward(char input) : KeyPair
Rotor1
- Rotor1() : Rotor
Rotor2
- Rotor2() : Rotor
Rotor3
- Rotor3() : Rotor
Rotor4
- Rotor4() : Rotor
Rotor5
- Rotor5() : Rotor

2.4. KeyPair Class

Represents a pairing of a left and right for character substitution within the rotor.

Attributes

- `rightKey:` The key representing the right side of the rotor.
- `leftKey:` The key representing the left side of the rotor.

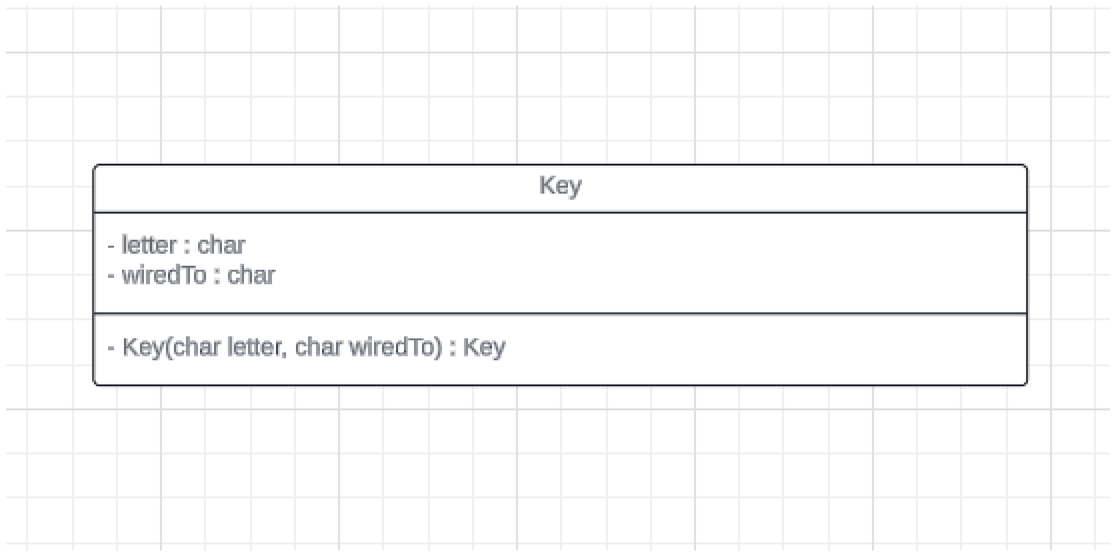


2.5. Key Class

This represents a single character in the rotor, including its original value and it's mapped value.

Attributes

- `letter:` The character itself
- `wiredTo:` The character it is wired for substitution.



3. Encryption Flow

The encryption process follows a structured sequence through the machine's components, starting with rotor rotation and ending with character substitution via the plugboard.

3.1. Forward Encryption

1. Rotor Rotation: Before processing each character, the rotors rotate based on their notch or after each key press (for rotor 3).
2. Plugboard substitution: The character is passed through the plugboard for initial substitution.
3. Rotor Encryption(Forward): The character passes through the rotors, starting from rotor 3 to rotor 1, where it is mapped from right to left in each rotor.

3.2. Reflector Substitution

Once the character reaches the reflector, it is mapped to a new character, ensuring that the encryption process is symmetrical.

3.3. Backward encryption

After reflector substitution, the character travels back through the rotors in reverse order, starting from rotors[0] to all the way to rotor[2] (rotor 3).

3.4.Final Plugboard Substitution

After passing through the rotors in backward, the character is passed through the plugboard once again for final substitution.

4. Plugboard functionality

The plugboard is pretty straightforward. Essentially, it allows the user to swap pairs of characters before and after the rotor encryption process. This extra layer of substitution makes the encryption even more secure. The configuration is user-defined and will be assigned as user inserts in the terminal.

5. Reflector

The reflector is the simplest thing in the whole architecture. Since it's symmetric, it makes the mapping much easier in terms of implementation. For example, if A maps to E through the reflector, E will map back to A.

6. Rotor rotation logic.

The rotors rotate after each keypress, with rotor 3 always rotating, Rotor 2 only rotates when rotor 3 reaches its notch, and rotor 1 rotates when rotor 2 reaches its notch. I have created dedicated methods both for atNotch() and rotate().