NES Encryptor

A Non-Encryption Standard Encryptor for the Enigma Machine

Authors

Tanvir Ahmed | Vanessa Miller | Kie Li | Vincente Prata

NJIT / July 2024

Table of Contents

[Class Diagram 4](#_Toc171284479)

[Context Diagram 5](#_Toc171284480)

[Descriptions 6](#_Toc171284481)

[Overall Subsystem 6](#_Toc171284482)

[General Overview 6](#_Toc171284483)

[Class Breakdown 6](#_Toc171284484)

[Initialization and Operation 7](#_Toc171284485)

[Use Cases 7](#_Toc171284486)

[Use Case I: Protection of Intellectual Property 7](#_Toc171284487)

[Use Case II: Secure Backup 8](#_Toc171284488)

[Classes 9](#_Toc171284489)

[ACSInterface Class 9](#_Toc171284490)

[Overview of ACSInterface Class 9](#_Toc171284491)

[Breakdown of ACSInterface Class 9](#_Toc171284492)

[Implementation of ACSInterface Class 9](#_Toc171284493)

[AASInterface Class 9](#_Toc171284494)

[Overview of AASInterface Class 9](#_Toc171284495)

[Breakdown of AASInterface Class 9](#_Toc171284496)

[Implementation of AASInterface Class 9](#_Toc171284497)

[Wheel Class 10](#_Toc171284498)

[Overview of Wheel class 10](#_Toc171284499)

[Breakdown of Wheel Class 10](#_Toc171284500)

[Implementation of Wheel Class 11](#_Toc171284501)

[WheelAssy Class 11](#_Toc171284502)

[Overview of WheelAssy Class 11](#_Toc171284503)

[Breakdown of WheelAssy Class 11](#_Toc171284504)

[Implementation of WheelAssy Class 12](#_Toc171284505)

[Reflector Class 12](#_Toc171284506)

[Overview of Reflector Class 12](#_Toc171284507)

[Breakdown of Reflector Class 12](#_Toc171284508)

[Implementation of Reflector Class 12](#_Toc171284509)

[Plugboard Class 13](#_Toc171284510)

[Overview of Plugboard Class 13](#_Toc171284511)

[Breakdown of Plugboard Class 13](#_Toc171284512)

[Implementation of Plugboard Class 13](#_Toc171284513)

[RedFileInterface Class 14](#_Toc171284514)

[Overview of RedFileInterface Class 14](#_Toc171284515)

[Breakdown of RedFileInterface Class 14](#_Toc171284516)

[Implementation of RedFileInterface Class 14](#_Toc171284517)

[BlackFileInterface Class 15](#_Toc171284518)

[Overview of BlackFileInterface Class 15](#_Toc171284519)

[Breakdown of BlackFileInterface Class 15](#_Toc171284520)

[Implementation of BlackFileInterface Class 15](#_Toc171284521)

[UserInput Class 15](#_Toc171284522)

[Overview of UserInput Class 15](#_Toc171284523)

[Breakdown of UserInput Class 16](#_Toc171284524)

[Implementation of UserInputClass 16](#_Toc171284525)

[NESEncryptorControl Class 16](#_Toc171284526)

[Overview of NESEncryptorControl Class 16](#_Toc171284527)

[Breakdown of NESEncryptorControl Class 16](#_Toc171284528)

[Implementation of NESEncryptorControl Class 17](#_Toc171284529)

[NESEncryptorFacade Class 18](#_Toc171284530)

[Overview of NESEncryptorFacade Class 18](#_Toc171284531)

[Breakdown of NESEncryptorFacade Class 18](#_Toc171284532)

[Implementation of NESEncryptorFacade Class 18](#_Toc171284533)

[Example Text 19](#_Toc171284534)

[Encryption 19](#_Toc171284535)

[Decryption 19](#_Toc171284536)

[Sequence Diagram 20](#_Toc171284537)

[Encryption 20](#_Toc171284538)

[Decryption 21](#_Toc171284539)

[Collaboration Diagram 22](#_Toc171284540)

[Encryption 22](#_Toc171284541)

[Decryption 23](#_Toc171284542)

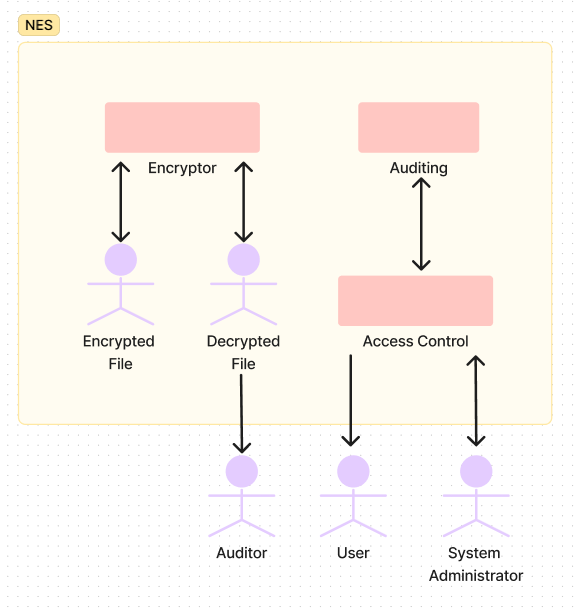
[Traceability Matrix 24](#_Toc171284543)

[Summary 24](#_Toc171284544)

# Class Diagram

# 

# Context Diagram



## 

# 

# 

# 

# Descriptions

## Overall Subsystem

### General Overview

The NESEncryptor subsystem encrypts and decrypts the input files it receives in a similar fashion to the infamous Enigma machine. Its structure mimics the rotor, or wheel, mechanism found in the physical Enigma machine in the form of Wheel classes.

### Class Breakdown

|  |  |  |
| --- | --- | --- |
| Class | Function | Singleton? |
| Wheel | simulates rotor mechanism used for encryption | no |
| WheelAssy | manages Wheels for unified character scrambling | yes |
| Reflector | “reflects” signal back through Wheels to complete encryption | yes |
| Plugboard | simulates “plugs” which swaps character if needed, providing further scrambling | yes |
| NESEncryptorControl | initiates and controls encryption/decryption | yes |
| NESEncryptorFacade | provides interface for encryption/decryption to coordinate between Plugboard, WheelAssy, and Reflector character at a time | yes |
| RedFileInterface | handles read/write operations for “red” plaintext file | yes |
| BlackFileInterface | handles read/write operations for “black” ciphertext file | yes |
| UserInput | takes user input to determine whether to encrypt or decrypt | yes |
| AASInterface | provides interface for status messages for encryption or decryption | yes |
| ACSInterface | performs authorization tasks | yes |

All classes will be described in depth throughout this document.

### Initialization and Operation

As the main entry point to the subsystem, NESEncryptorControl initializes all the classes. This is possible because all classes, except the Wheels, have been instantiated using the Singleton Design Pattern to ensure that each one has only one instance throughout the application. The Wheels are an exception because there can be more than one, increasing the complexity to scramble the character transformation.

After successful initialization, operation begins and follows the order of events below, if there is no errors:

1. The main function calls the NESEncryptorControl to start the process
2. ACSInterface authorizes the process.
3. UserInput prompts the user to choose between encryption or decryption.
4. Input is handled depending on user choice. It uses the RedFileInterface for reading if the user selects encryption, and the BlackFileInterface for reading if the user selects decryption.
5. NESEncryptorFacade oversees the encryption or decryption process by coordinating Plugboard, WheelAssy, and Reflector to simulate the “turning” of the Wheels to get each encrypted character.
6. AASInterface prints status messages regarding the success or failure of the encryption or decryption process.

### Use Cases

The original use of the Enigma machine was for the German military to send and receive encrypted messages during World War II. This NESEncryptor subsystem, modeled after the mechanisms of the Enigma Machine, could also have use cases for secure communication in military operations. However, in this NESEncryptor subsystem, the range of characters has been expanded from only alphabets to ASCII characters, broadening its use case and complexity.

#### Use Case I: Protection of Intellectual Property

**Actors:** Healthcare Data Security

**Description:** Healthcare Data Security deals with materials that often contain sensitive medical information about patients that must be secure. The NESEncryptor subsystem would encrypt these documents to prevent unauthorized access to the information that they contain.

**Steps:**

1. An employee of an R&D department chooses a document containing intellectual property to secure.
2. The NESEncryptorControl is initiated.
3. Upon being prompted by UserInput, the employee chooses to encrypt.
4. The plaintext document is read by RedFileInterface.
5. NESEncryptorFacade coordinates the process of encrypting the contents of the document.
6. BlackFileInterface writes the encrypted document.
7. The encrypted document can now be stored securely.

#### Use Case II: Secure Backup

**Actors:** System administrators

**Description:** System administrators often encounter sensitive data that needs to be securely backed up so that it is both recoverable and protected. The NESEncryptor subsystem can ensure that data is encrypted before it is backed up.

**Steps:**

1. A system administrator chooses some data to be backed up.
2. The NESEncryptorControl is initiated.
3. Upon being prompted by UserInput, the system administrator chooses to encrypt.
4. The plaintext document is read by RedFileInterface.
5. NESEncryptorFacade coordinates the process of encrypting the contents of the document.
6. BlackFileInterface writes the encrypted data.
7. The encrypted data can now be stored securely as a backup.

## Classes

### ACSInterface Class

#### Overview of ACSInterface Class

The ACSInterface class defines an interface which handles authorization.

#### Breakdown of ACSInterface Class

ACSInterface is implemented through a private constructor and a public static Instance() method to maintain the Singleton Design Pattern.

The inline method authorize() relies on std::cout to print authorization messages to the console.

#### Implementation of ACSInterface Class

The header file ACSInterface.h:

* defines the ACSInterface class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public method authorize().

### AASInterface Class

#### Overview of AASInterface Class

The AASInterface class defines an interface which sends the status messages for encryption and decryption.

#### Breakdown of AASInterface Class

AASInterface is implemented through a private constructor and a public static Instance() method to maintain the Singleton Design Pattern.

The inline methods sendEncryptOK() and sendDecryptOK() rely on std::cout to output success or failure messages.

#### Implementation of AASInterface Class

The header file AASInterface.h:

* defines the AASInterface class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods:
  + sendEncryptOK()
  + sendDecryptOK()

### Wheel Class

#### Overview of Wheel class

The Wheel class is a simulation of the rotor mechanism used in the Enigma machine which provides the encryptor with bi-directional value translation, allowing for both encryption and decryption.

#### Breakdown of Wheel Class

Wheel is initialized with:

* number of entries nEnts
* a random seed to ensure variation between each instantiation
* a starting wheel position cPos.

Two vectors are also initialized with random values to ensure variation:

* lToR, which stores left-to-right transformations
* rToL, which stores right-to-left transformations.

Offset conversion is implemented through the convertToOffset method where the offsets can be stored according to their indices.

Once translation begins, two methods are employed incorporating use of the vectors mentioned above:

* getLtoR(short i), which performs a left-to-right translation using the given index ‘i’ as its parameter
* getRtoL(short i),which performs a right-to-left translation using the given index ‘i’ as its parameter

After the whole process of randomization, conversion, and transformation is completed, the method advance() can be used to advance the position of the wheel, or reversed using the reverse() method. To reset the position of the wheel the method resetCur(short cP) is used in which short cP is the value of the wheel’s current position.

Finally, the atNotch() method is used to check if the current wheel position matches the notch position.

#### Implementation of Wheel Class

The header file Wheel.h:

* defines the Wheel class with the above specifications.
* includes public methods such as constructors, destructors, and inline functionality.

The source file Wheel.cpp:

* implements and initializes the Wheel constructor.
* fills the vectors with random values by using loadLtoR and loadRtoL.
* adjusts the offset of the vectors by using convertToOffset.
* handles transformation via the getLtoR and getRtoL methods.
* clears the transformation vectors at the end of the encryption/decryption process.

### WheelAssy Class

#### Overview of WheelAssy Class

The WheelAssy class puts together the three Wheel objects that are used in the encryptor. It utilizes the Singleton Design Pattern to unify encryption and decryption across all three Wheels.

#### Breakdown of WheelAssy Class

The WheelAssy class is a Singleton through the static Instance() method to ensure that there only exists one instance of WheelAssy in the system.

The three wheels, wheel1, wheel2, and wheel3, are initialized through the methods described in the Wheel class breakdown. Each wheel can be advanced through the advance() method.

Repeating the functions of the Wheel class, WheelAssy provides bi-directional translation through the methods lToR and rToL and their respective methods from the Wheel class, getLtoR and getRtoL. All three wheels can be reset to their original positions through the reset() function as a means of restarting the wheel assembly altogether.

#### Implementation of WheelAssy Class

The **header** file WheelAssy.h:

* defines the Wheel class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods.

The source file WheelAssy.cpp:

* implements and initializes the WheelAssy constructor.
* advances the wheels sequentially through the advance() method.
* handles transformation via the lToR and rToL methods.
* reinitializes the wheels to their original positions via the reset() method.
* deletes dynamically allocated Wheel objects at the end of the encryption/decryption process.

### Reflector Class

#### Overview of Reflector Class

The Reflector class “reflects” input values by mapping them to their corresponding output values. Their relationship is defined by a set definition of mappings. It utilizes the Singleton Design Pattern.

#### Breakdown of Reflector Class

Similar to WheelAssy, Reflector is implemented through a private constructor and a static Instance() method to maintain the Singleton Design Pattern.

The mapping of input and output values is performed through initialization of an array numbers. An acceptable range of input values is defined by OFFSET and LAST, and any input character that is out of the acceptable range will print an error message and terminate the process. Otherwise, the reflect() method will map an input value to its output value according to the mapping provided by numbers.

#### Implementation of Reflector Class

The **header** file Reflector.h:

* defines the Reflector class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods.

The source file Reflector.cpp:

* implements and initializes the Reflector constructor.
* initializes the numbers array and its mappings.
* utilizes the mappings to reflect() input values while performing range validation.

### 

### Plugboard Class

#### Overview of Plugboard Class

The Plugboard class uses a predefined set of plug connections to map some specific input values to alternative output values. It also uses the Singleton Design Pattern.

#### Breakdown of Plugboard Class

Plugboard is implemented through a private constructor and a static Instance() method to maintain the Singleton Design Pattern.

The mapping of specific characters is done through initialization of plugs, which is a map of int.

Range validation is performed via the getPBC() method using OFFSET and LAST to define the acceptable input range. An input character outside this range will print an error message and terminate the program. Otherwise, getPBC() will check that a valid input character has a predefined output value in plugs.

* If the mapping exists, getPBC() returns the mapped character.
* If the mapping does not exist, getPBC() returns the original character.

#### Implementation of Plugboard Class

The **header** file Plugboard.h:

* defines the Plugboard class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods.

The source file Plugboard.cpp:

* implements and initializes the Reflector constructor.
* initializes the plugs map and its mappings.
* utilizes the mappings in the getPBC() method to transform characters while also performing range validation.

### RedFileInterface Class

#### Overview of RedFileInterface Class

RedFileInterface defines an interface used to read from and write to the “red” plaintext file.

#### Breakdown of RedFileInterface Class

RedFileInterface is implemented through a private constructor and a public static Instance() method to maintain the Singleton Design Pattern.

Each function in RedFileInterface returns a boolean value. The returned value indicates if the function was successful or not.

#### Implementation of RedFileInterface Class

The **header** file RedFileInterface.h:

* defines the RedFileInterface class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods:
  + openForRead() and openForWrite()
  + getNextChar() and putNextChar()
  + eof() and close()

The **source** file RedFileInterface.cpp:

* opens “MyInFile.txt” for reading using the method openForRead().
* opens “MyOutFile.txt” for reading using the method openForWrite().
* gets the next character from the file using getNextChar().
* writes a character to the file using putNextChar(unsigned char c).
* checks if the end of the file has been reached with eof().
* closes the file with close().

### BlackFileInterface Class

#### Overview of BlackFileInterface Class

The BlackFileInterface class defines an interface for reading and writing files, similar to RedFileInterface. However, it instead deals with the “black” ciphertext file.

#### Breakdown of BlackFileInterface Class

BlackFileInterface is implemented through a private constructor and a public static Instance() method to maintain the Singleton Design Pattern.

Just like RedFileInterface, function in BlackFileInterface returns a boolean value. The returned value indicates if the function was successful or not.

#### Implementation of BlackFileInterface Class

The **header** file BlackFileInterface.h:

* defines the BlackFileInterface class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods:
  + openForRead() and openForWrite()
  + getNextChar() and putNextChar()
  + eof() and close()

The source file BlackFileInterface.cpp:

* opens “MyOutFile.dat” for reading using the method openForRead().
* opens “MyOutFile.dat” for reading using the method openForWrite().
* gets the next character from the file using getNextChar().
* writes a character to the file using putNextChar(unsigned char c).
* checks if the end of the file has been reached with eof().
* closes the file with close().

### UserInput Class

#### Overview of UserInput Class

The UserInput class deals with the input received from the user in order to determine whether to move forward with encryption or decryption.

#### Breakdown of UserInput Class

UserInput is implemented through a private constructor and a public static Instance() method to maintain the Singleton Design Pattern.

Only two inputs are accepted from the user by UserInput. An enumerated Action is defined with the two possible paths being encryption (ENC) or decryption (DEC). The user is prompted to enter this information by the method getUserAction().

Should the user enter some unrecognized input as the Action, the program will print an error message and terminate.

#### Implementation of UserInputClass

The header file UserInput.h:

* defines the UserInput class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods:
  + Action
  + getUserAction()

The source file UserInput.cpp:

* implements the method getUserAction() to prompt the user to either type ‘e’ for encryption or ‘d’ for decryption, and then reads the user input and executes the Action value.

### NESEncryptorControl Class

#### Overview of NESEncryptorControl Class

Effectively, NESEncryptorControl is the source of centralized control of the entire system. It coordinates interaction between the different subsystems of the NES Encryptor. As such, it is implemented as a Singleton to ensure only one point of control.

#### Breakdown of NESEncryptorControl Class

NESEncryptorControl is implemented through a private constructor and a public static Instance() method to maintain the Singleton Design Pattern.

Subsystem integration is performed, referencing the following classes via the single points of access due to their Singleton pattern:

* ACSInterface
* AASInterface
* UserInput
* RedFileInterface
* BlackFileInterface
* EncryptorFacade

The start() method authorizes the user through ACSInterface. If authorization fails, the program is terminated.

NESEncryptorControl then enters a loop which asks the user for input via UserInput to determine whether to proceed with

* encryption, which reads from “red” to “black” (via RedFileInterface and BlackFileInterface), or
* decryption, which reads from “black” to “red” and decrypts the encrypted file.

Actual encryption and decryption is then redirected to NESEncryptorFacade.

#### Implementation of NESEncryptorControl Class

The header file NESEncryptorControl.h:

* defines the constructors of ACSInterface, AASInterface, UserInputs, RedFileInterface, BlackFileInterface, and NESEncryptorFacade class.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public method:
  + start()

The source file NESEncryptorControl.cpp:

* implements the start() method to commence authorization, user input, and encryption/decryption.
* continuously loops to handle user actions.
* Reads the file with openForRead() until end of file using eof()
* Gets the next char using getNextChar()
* Calls NESEncryptorFacade to to encrypt the character
* Put the char in another file using putNextChar()
* At last close the files using close() method file interfaces
* After all the characters are encrypted, a success status is sent using AAS.sendEncryptOK() and AAS.sendDecryptOK() for decrypt
* Repeats the same thing for decryption

### NESEncryptorFacade Class

#### Overview of NESEncryptorFacade Class

NESEncryptorFacade primarily serves to handle the actual encryption and decryption of the NES Encryptor. It does this by directing the interactions between the Singleton classes which participate in encryption/decryption. It is implemented as a Singleton as well.

#### Breakdown of NESEncryptorFacade Class

NESEncryptorFacade is implemented through a private constructor and a public static Instance() method to maintain the Singleton Design Pattern.

An input character provided by the user input is transformed through the encrypt() function. The character follows the path below:

1. Transformation through the methods in Plugboard.
2. Transformation from right to left in WheelAssy.
3. Reflection in Reflector corresponding to the mapping of its output value.
4. Transformation again through WheelAssy, but this time from left to right.
5. Transformation again in Plugboard.

After this encryption process is complete, WheelAssy advances the wheel.

The function decrypt() follows the above five stages of transformation, but in reverse order to obtain the original input character.

At the end of either encryption or decryption, WheelAssy is reset to its original state by NESEncryptorFacade through the reset() method.

#### Implementation of NESEncryptorFacade Class

The header file NESEncryptorFacade.h:

* defines the NESEncryptorFacade class with the above specifications.
* has a private constructor to maintain the Singleton Design Pattern.
* provides access to the single instance resulting from the Singleton pattern via the static Instance() method.
* includes the public methods:
  + encrypt(unsigned char& c)
  + decrypt(unsigned char& c)
  + reset()

The source file NESEncryptorFacade.cpp:

* implements the encrypt() method to encrypt a valid input character going through Plugboards, WheelAssy, Reflector adjusting the character in the process.
* Implements the decrypt() method to reverse encryption.
* implements reset() to reset WheelAssy.

# Example Text

#### Encryption

1. Input text (MyInFile.txt) to be encrypted:

Ex: “ Mary had a little lamb”.

1. Upon UserInput class prompt, type ‘e’ to encrypt
2. Encryption process occurs
3. View MyOutFile.dat to see encrypted text:

Ex: “$ IqGHs<w-h61`<N"!4tELt”

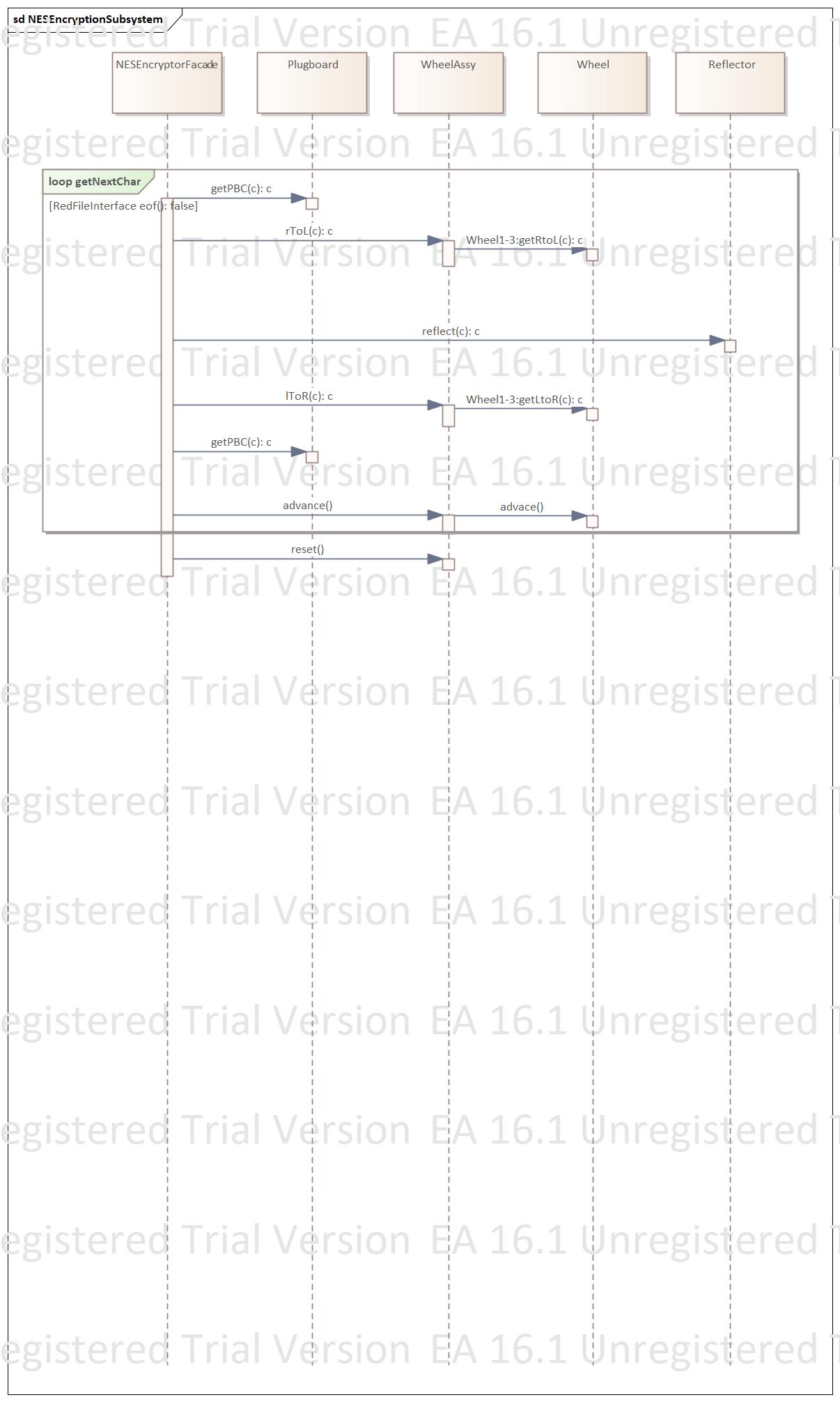
#### Decryption

1. Input text (MyOutFile.dat) to be decrypted:
2. Upon UserInput class prompt, type ‘d’ to decrypt
3. Decryption process occurs
4. View MyOutfile.txt to see decrypted text:

Ex: “ Mary had a little lamb”.

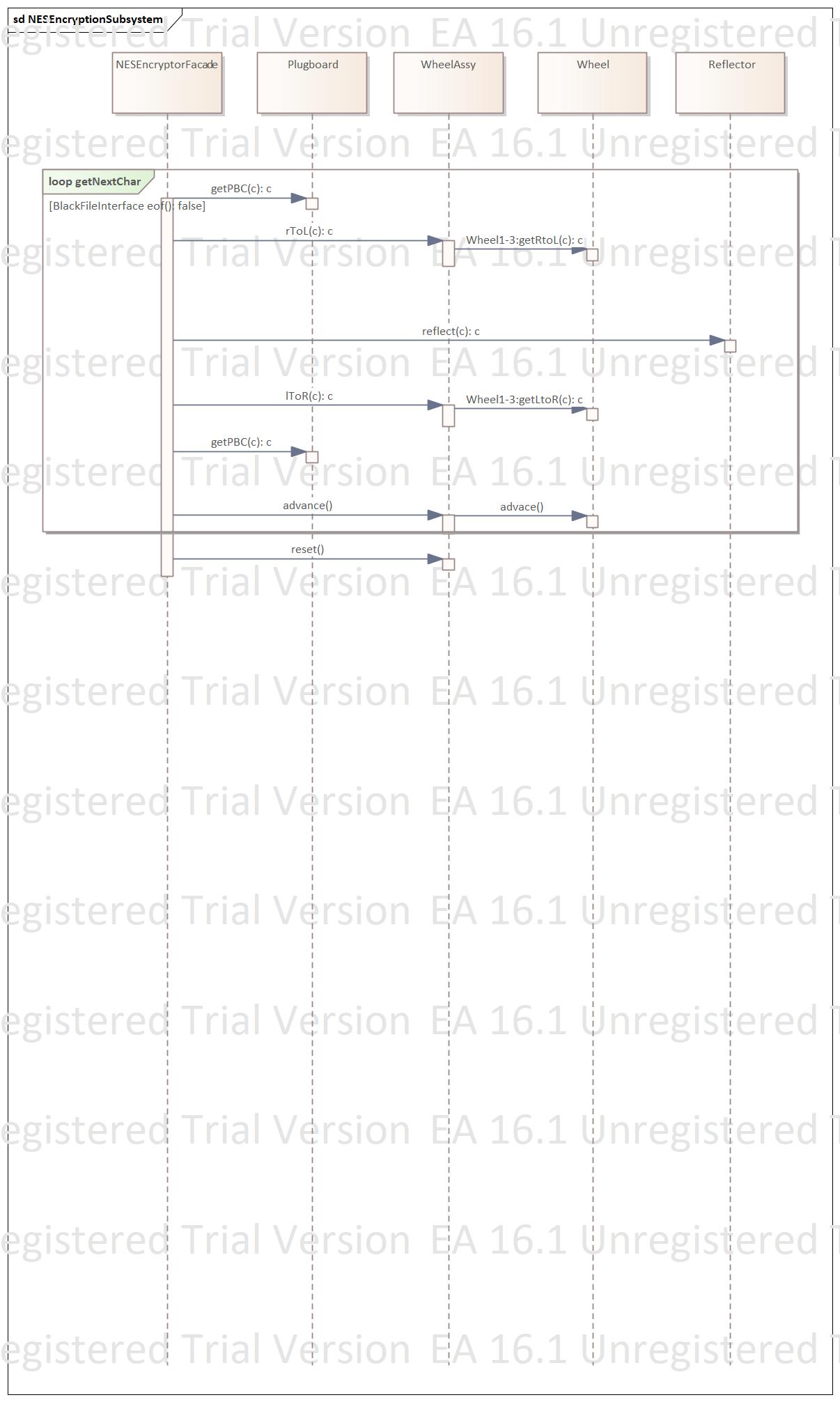
# Sequence Diagram

### Encryption



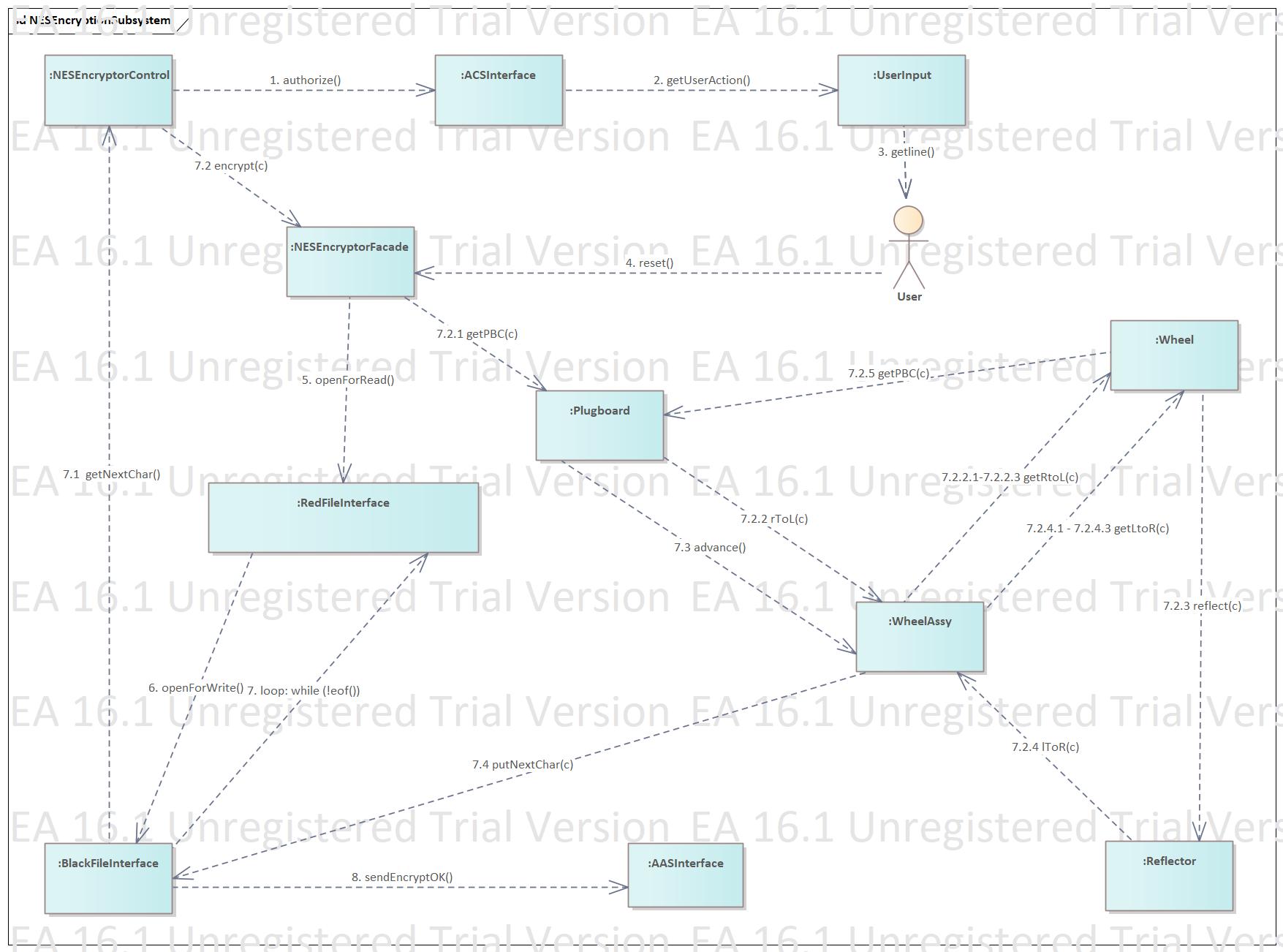
# 

### Decryption



# Collaboration Diagram

## Encryption



## Decryption

# Traceability Matrix

|  |  |  |
| --- | --- | --- |
| RAD Element | Type | Implemented Classes |
| AccessControl | Entity | AASInterface |
| AuditAndAlert | Boundary | ACSInterface |
| User | Boundary | UserInput |
| NESEncryptor\_ | Control | NESEncryptorControl |
| NESEncryptor | Entity | NESEncryptorFacade  WheelAssy  Plugboard  Reflector |
| EncryptedFile | Boundary | BlackFileInterface |
| DecryptedFile | Boundary | RedFileInterface |

## 

# Summary

This NESEncryptor subsystem provides a reliable method for converting data between plaintext and ciphertext. Modeled after the rotor mechanisms of the Enigma machine, it ensures robust encryption and decryption. The architecture employs the Singleton Design Pattern, ensuring consistent encryption and decryption processes across the system.