UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Elaborato Ingegneria del Software

Data Lock, applicativo java adibito alla cifratura di dati su terminali accessibili ad utenti con livelli differenti di accesso

Piero Turri

A.A. 2019-2020

Indice

Introduzione

Motivazione e Contenuti

L'elaborato consiste nello sviluppo di un'applicazione, scritta in linguaggio Java, per aumentare il livello di sicurezza dei dati salvati sul computer (e.g. credenziali di accesso o dati sensibili).
L'applicazione, Data Lock, è pensata per essere installata su terminali accessibili a due tipologie di utenti: un utente base e un utente admin. Entrambi gli utenti avranno la possibilità di utilizzare a pieno tutte le funzionalità dell'applicazione, in più l'utente admin avrà la possibilità di visionare un file di log, in cui verranno salvati tutti le operazioni eseguite da Data Lock.

Attraverso una schermata di login l'utente potrà accedere al pannello principale dell'applicazione dove potrà decidere se caricare un testo, già presente sul terminale in uso e salvato su un file con formato .txt, oppure scrivere un nuovo testo da cifrare per aumentarne la sicurezza.
Una volta inserito il testo nell'apposito spazio, attraverso uno dei due metodi sopra elencati, l'utente sceglierà la modalità di cifratura che reputa migliore per i dati che vuole mettere in sicurezza. Gli algoritmi forniti dall'applicazione per cifrare il testo sono: l'algoritmo di cifratura AES e il cifrario di Cesare.
Dopodiché l'utente cifrerà il messaggio o lo decifrerà, in base al suo volere, per poi andare a salvare il file. Al momento del salvataggio, attraverso un'apposita schermata, l'utente potrà vedere come si chiama il nuovo file da lui creato, oppure potrà accertarsi che il file, da lui caricato in precedenza, è stato sovrascritto con successo. Inoltre, nella schermata di salvataggio verrà visualizzata la chiave di cifratura utilizzata per cifrare/decifrare il testo.
Una volta chiusa la schermata di salvataggio, tutti i campi del pannello principale di Data Lock verranno resettati per permettere all'utente di cifrare o decifrare un nuovo testo.

Quando l'admin farà l'accesso, vedrà nel pannello principale un'icona di un ingranaggio, su cui poter cliccare per poter accedere alla sezione riepilogativa dei processi eseguiti da Data Lock. In questo pannello l'admin potrà vedere quale utente ha eseguito un processo specifico, quale file è stato creato/modificato durante quel processo, la data e l'ora in cui l'utente ha effettuato il processo. I dati riguardanti i processi eseguiti dall'applicazione vengo costantemente registrati, su un file salvato sul terminale in uso, nel momento in cui l'utente, admin o base, decide di salvare un testo. Nel momento in cui l'admin entra nella sua area riservata, l'applicazione andrà a caricare il file utilizzato come registro e lo mostrerà nell'area apposita.

Siccome l'applicazione è pensata per essere installata su un terminale a cui possono accedere due diversi tipi di utenti, è fornita di un sistema di log out in modo da dare la possibilità di essere lasciata aperta invece di essere chiusa dopo ogni utilizzo.

Modalità di sviluppo

Per la realizzazione di Data Lock è stato utilizzato il linguaggio di programmazione Java attraverso l'IDE Eclipse. Nella fase di progettazione e nella fase di realizzazione dell'interfaccia grafica, sono stati identificati i casi d'uso e rappresentati attraverso uno use case diagram. Inoltre, è stato sviluppato anche un sequence diagram per un possibile funzionamento dell'applicazione
È stata definita e realizzata una logica di dominio in prospettiva di specifica e di implementazione attraverso un diagramma UML.

Per lo sviluppo del programma sono stati adoperati alcuni pattern comportamentali come il pattern Observer tra la classe che salva il file e la classe che modifica il file di log, due pattern Strategy usati rispettivamente per andare a settare la strategia per cifrare i dati e per settare la strategia per decifrare i dati a seconda di cosa vuole fare l'utente. Inoltre, sono state usate due varianti differenti del pattern Proxy: è stato usato il Protection Proxy per la gestione degli accessi all'applicazione, mentre per la gestione del caricamento dei file è stato usato il Virtual Proxy.

Per quanto riguarda l'interazione con l'utente finale è stata realizzata una GUI (Graphic Unit Interface) attraverso il framework Java Swing adottando il pattern strutturale MVC (Model View Controller) dove il model è rappresentato attraverso la domain logic contenuta nel package model mentre la business logic è contenuta sia nel model che nel controller.

Per l'implementazione dell'algoritmo di cifratura AES sono stati adoperati il package javax.crypto e il package java.security per la generazione delle chiavi simmetriche adoperate poi per la cifratura dei messaggi.

Una parte fondamentale è data anche dalla realizzazione di alcune classi di testing (più precisamente di Unit Testing) attraverso il framework JUnit 5 integrato direttamente nell' ambiente di sviluppo.

L'intero progetto è stato versionato con Git ed è disponibile su GitHub al seguente link:
https://github.com/PieTurri/DataLock

Casi d’uso

Dopo aver definito il problema in questione sono stati individuati gli attori in gioco e i vari casi d’uso.

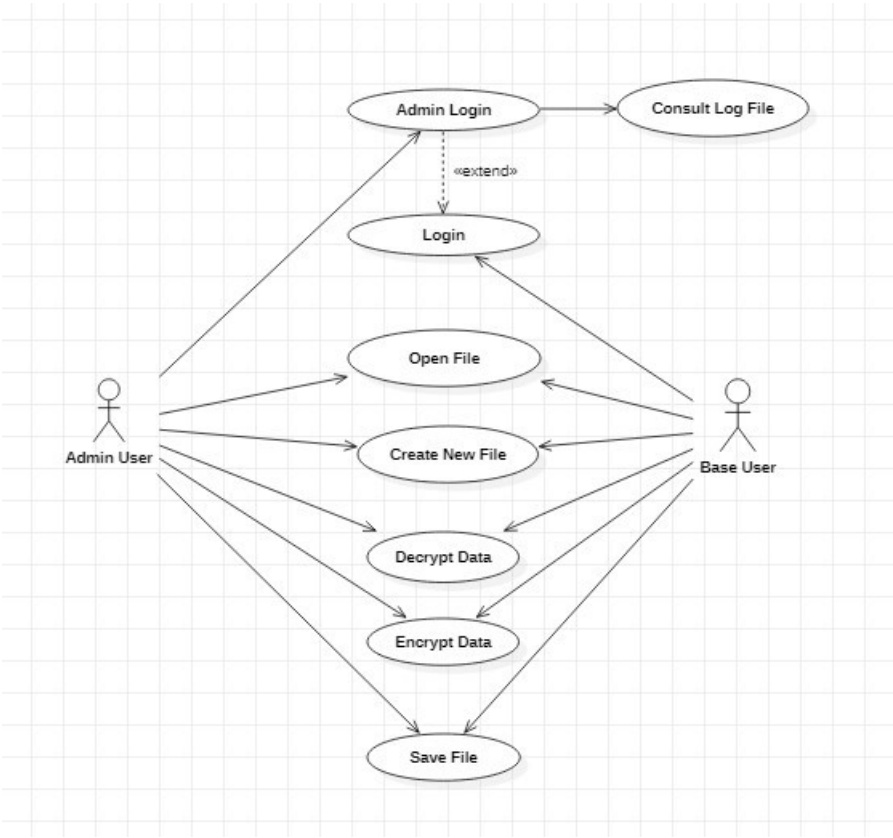


Figura 1 – Diagramma dei casi d’uso

↳ Diagramma UML

Qui di seguito la realizzazione del diagramma UML che descrive la logica di dominio in prospettiva di implementazione.

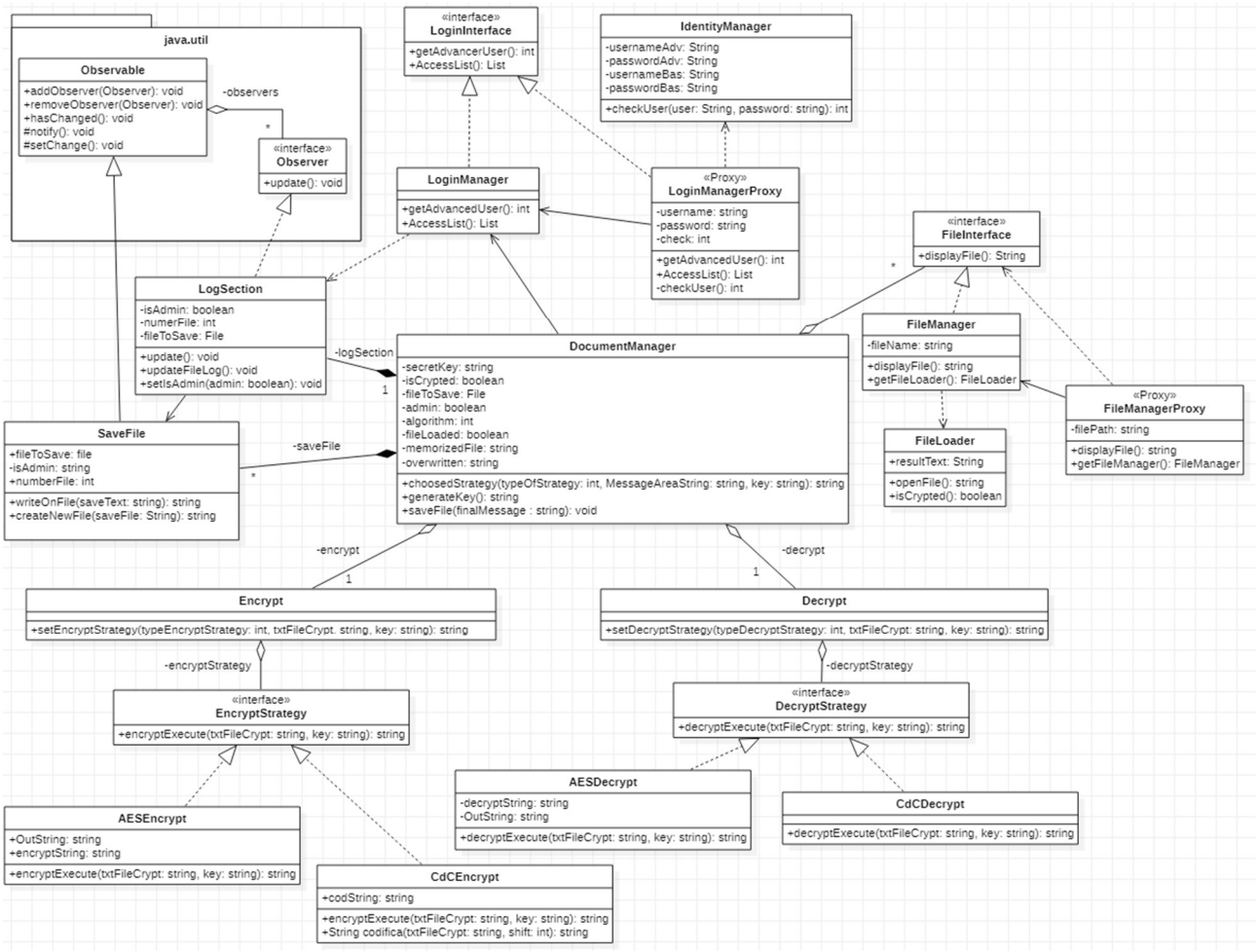
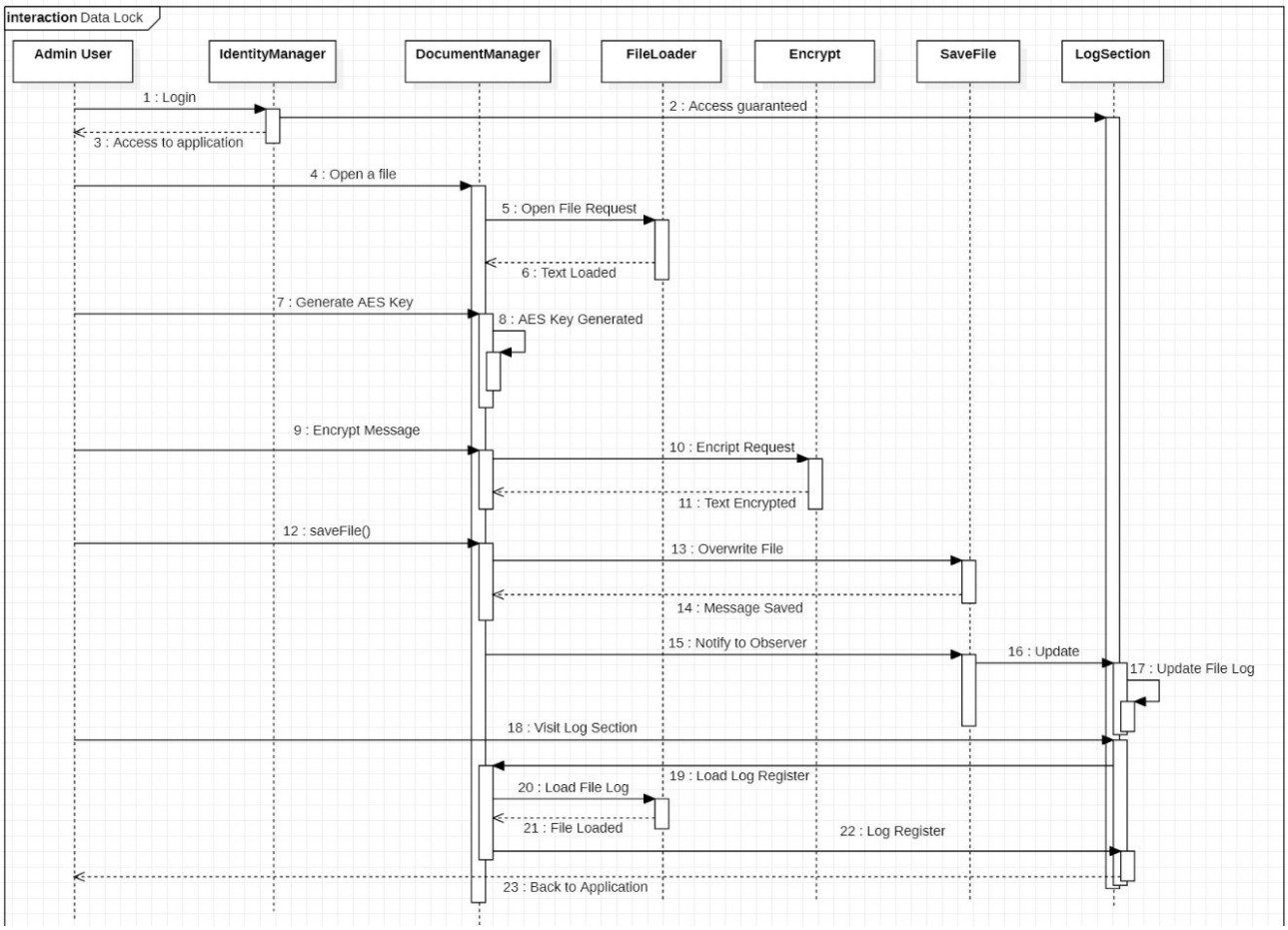


Figura 2 - Diagramma UML

↳ Sequence Diagram

Qui di seguito un Sequence Diagram che simula un possibile flusso d'esecuzione del programma con l'utente admin che dopo aver cifrato un testo caricato dal terminale, visiona i processi effettuati dall'applicazione.



Implementazione Grafica - Mockups

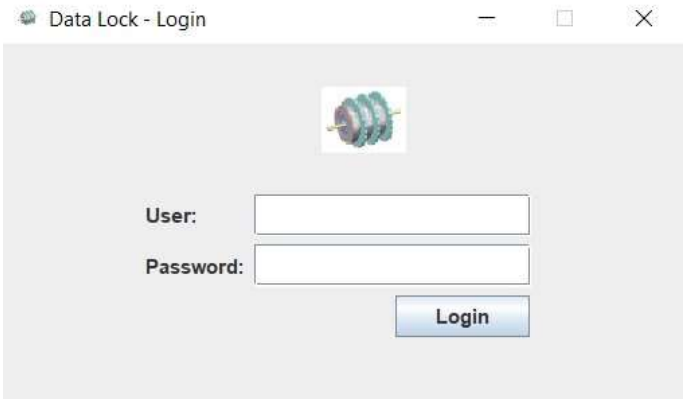


Figura 3 - Schermata Login

Figura 4 - Pannello Principale Utente Admin

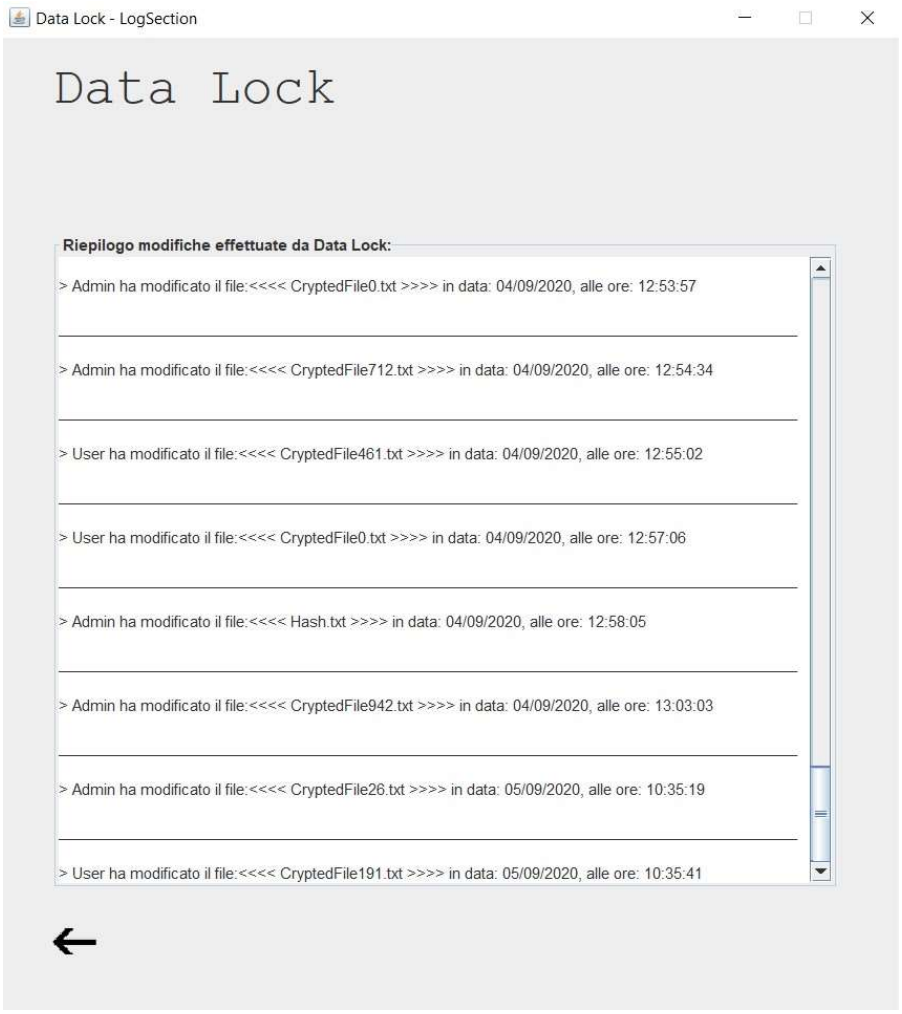


Figura 5 - Pannello Riepilogativo Processi Effettuati Data Lock

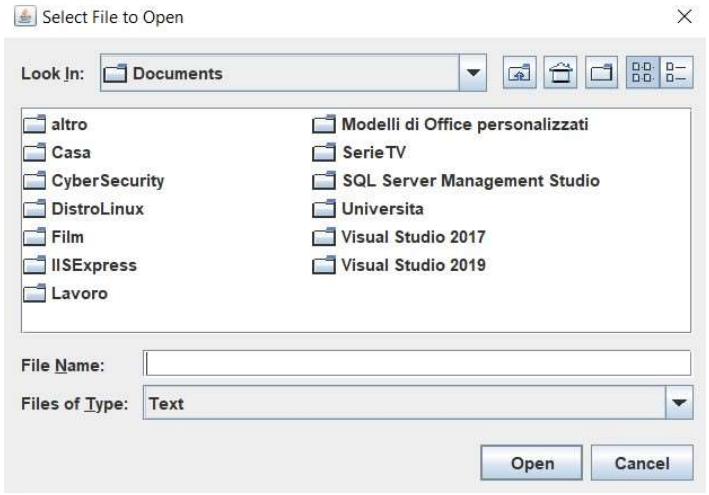


Figura 6 - Pannello Caricamento File

Figura 7 - Salvataggio Nuovo File

Figura 8 - Sovrascrittura File

Figura 9 - Errore Inserimento Chiave AES

Figura 10 - Errore Messaggio non decifrabile: Quando viene usato il cifrario di Cesare per cifrare testi con caratteri speciali

Figura 11 - Pannello Principale Utente Base

Classi ed Interfacce

Per l'implementazione sono state definite nuove classi ed interfacce o riutilizzate quelle della libreria standard Java (e.g. le classi e le interfacce contenute nel package java.util). Le principali classi dell'applicazione contenute nel package **model** sono:

- IdentityManager
- DocumentManager
- LogSection
- AESEncrypt
- AESDecrypt
- CdCEncrypt
- CdCDecrypt
- FileLoader
- SaveFile

Oltre a queste classi principali ne sono state definite altre sia all'interno del package **model** sia all'interno degli altri due package presenti in src (il package **controller** e il package **view**).

In particolare, sono state definite varie classi view, che estendendo JFrame(dal framework Java.swing), per la parte grafica dell'applicativo (contenute nel package **view**).

1. IdentityManager

La classe IdentityManager rappresenta il fulcro su cui si articola il pattern Protection Proxy. Al suo interno possiamo trovare i parametri di accesso all'applicazione, che nel nostro caso sono statici e l'accesso è consentito solo a due tipi di utente, ma questa classe potrebbe dialogare con un database per consentire un accesso dinamico.

```
package model;

public class IdentityManager {

    private static String usernameAdv = "root";
    private static String passwordAdv = "toor";
    private static String usernameBas = "user";
    private static String passwordBas = "password";

    public static int checkUser(String user, String pass) {
        if(usernameAdv.equals(user) && passwordAdv.equals(pass))
            return 0;
        else if (usernameBas.equals(user) && passwordBas.equals(pass))
            return 1;
        else
            return 2;
    }
}
```

Figura 12 - Classe IdentityManager

2. DocumentManager

La classe DocumentManager è il cuore dell'applicazione. Questa permette all'utente di svolgere tutte le operazioni che Data Lock fornisce, come caricare un testo da decifrare o da cifrare, creare un nuovo testo da cifrare, scegliere l'algoritmo di cifratura, inserire una chiave di cifratura a propria scelta oppure generarne una nuova, ed infine salvare il testo sul terminale.

```
public String openFile(String absolutePath) {

    FileManagerProxy = new FileManagerProxy(absolutePath);

    return FileManagerProxy.displayFile();
}
```

Figura 13 - Metodo per il caricamento di un testo

```
public String choosedStrategy(int typeOfStrategy, String MessageAreaString, String key) throws Exception{
    System.out.println("Message: " + MessageAreaString + " Type of Strategy: " + typeOfStrategy +" Key: " + key);
    switch (typeOfStrategy) {
        case 0: {
            return encrypt.setEncryptStrategy(algorithm, MessageAreaString, key);

        }case 1: {
            return decrypt.setDecryptStrategy(algorithm, MessageAreaString, key);
        }
        default: return null;
    }
}
```

Figura 14 - Metodo per settare la strategia scelta dall'utente

```java
public String generateKey(){
    switch (getAlgorithm()) {
        case 0: {
            String AlphaNumericString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  + "abcdefghijklmnopqrstuvxyz";
            StringBuilder sb = new StringBuilder(16);

            for (int i = 0; i < 16; i++) {
                int index = (int)(AlphaNumericString.length()* Math.random());
                sb.append(AlphaNumericString.charAt(index));
            }

            setSecretKey(new String(sb));
            return getSecretKey();
        }
        case 1: {

            int n = -1;
            boolean flag = false;
            while(!flag) {
                n = (int) Math.floor(Math.random() * 26);
                if(n < 26)
                    flag = true;
            }
            setSecretKey(new String(Integer.toString(n)));
            return getSecretKey();
        }
        default: return null;
    }
}
```

Figura 15 - Metodo per la generazione di una chiave di cifrario

```java
public void saveFile(String finalMessage) throws Exception {

    if (getFileLoaded()) {
        saveFile.setAdmin(getAdmin());
        saveFile.fileToSave = getFileToSave();

        setMemorizedFile(saveFile.writeOnFile(finalMessage));

        logSection.setFileName(saveFile.fileToSave);
        logSection.setNumberFile(saveFile.getNumberFile());
        logSection.setIsAdmin(getAdmin());

        saveFile.notifyObservers();

        setOverwritten(true);
    } else {
        saveFile.setAdmin(getAdmin());
        saveFile.fileToSave = null;

        setMemorizedFile(saveFile.createNewFile(finalMessage));

        logSection.setFileName(saveFile.fileToSave);
        logSection.setNumberFile(saveFile.getNumberFile());
        logSection.setIsAdmin(getAdmin());

        saveFile.notifyObservers();

        setOverwritten(false);
    }
}
```

Figura 16 - Metodo per salvare i testi modificati dall'utente

3. LogSection

La classe LogSection ha una natura bifunzionale. Infatti, è parte integrante sia del pattern Protection Proxy, sia del pattern Observer. In particolare, questa classe implementando il metodo update() del pattern Observer va a sovrascrivere i dati contenuti all'interno del file CryptoFileLog.txt . Questo permette poi di visionare, nella LogSectionView accessibile solo all'admin, i processi effettuati da DataLock.

```
@Override
public void update(Observable o, Object arg) { updateFileLog(); }

private void updateFileLog() {

    File fileLog= new File("C:/Users/piero/Desktop/CryptoFileLog.txt");

    fileLog.setWritable(true);
    FileOutputStream fileOutputStream = null;
    try {
        fileOutputStream = new FileOutputStream("C:/Users/piero/Desktop/CryptoFileLog.txt",true);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd/MM/yyyy");
    DateTimeFormatter dtf1 = DateTimeFormatter.ofPattern("HH:mm:ss");
    LocalDateTime now = LocalDateTime.now();

    if(isAdmin) {

        if(fileLog.exists()) {
            PrintWriter scriviPrintWriter = new PrintWriter(fileOutputStream); //serve per creare una variabile che immagazzina il testo da scrivere
            scriviPrintWriter.append("\n");
            scriviPrintWriter.append("\n");
            scriviPrintWriter.append("_____________________________________________________________________________");
            scriviPrintWriter.append("\n");
            scriviPrintWriter.append("\n");
            if(fileToSave == null) {
                scriviPrintWriter.append("> Admin ha modificato il file:<<<< CryptedFile"+ getNumberFile() +".txt >>>> in data: " + dtf.format(now) + ", alle ore: " +  dtf1.format(now));
            }
            else
                scriviPrintWriter.append("> Admin ha modificato il file:<<<< " + fileToSave.getName() + " >>>> in data: " + dtf.format(now) + ", alle ore: " +  dtf1.format(now));
            scriviPrintWriter.close();
            fileLog.setReadOnly();
        }
    }else {

        if(fileLog.exists()) {
            PrintWriter scriviPrintWriter = new PrintWriter(fileOutputStream); //serve per creare una variabile che immagazzina il testo da scrivere
            scriviPrintWriter.append("\n");
            scriviPrintWriter.append("\n");
            scriviPrintWriter.append("_____________________________________________________________________________");
            scriviPrintWriter.append("\n");
            scriviPrintWriter.append("\n");
            if(fileToSave == null) {
                scriviPrintWriter.append("> User ha modificato il file:<<<< CryptedFile"+numberFile+".txt >>>> in data: " + dtf.format(now) + ", alle ore: " +  dtf1.format(now));
            }
            else
                scriviPrintWriter.append("> User ha modificato il file:<<<< " + fileToSave.getName() + " >>>> in data: " + dtf.format(now) + ", alle ore: " +  dtf1.format(now));
            scriviPrintWriter.close();
            fileLog.setReadOnly();
        }
    }
    boolean flag = fileLog.setReadOnly();
    if (flag==true)
    {
       System.out.println("File successfully converted to Read only mode!!");
    }
    else
    {
      System.out.println("Unsuccessful Operation!!");
    }
}
```

Figura 17 - Override metodo update()

4. AESEncrypt

La classe AESEncrypt serve per cifrare un testo con l'algoritmo AES.

AES (Advanced Encryption Standard) è un algoritmo di cifratura a blocchi[1]. Si tratta di un algoritmo di cifratura a chiave simmetrica[2] che ha la caratteristica di cifrare un blocco di bit contemporaneamente. È ampiamente utilizzato sia per sviluppi hardware che software, ha

[1] Algoritmo di cifratura a blocchi: è un algoritmo a chiave simmetrica (o a chiave privata) che opera su un gruppo di bit di lunghezza finita organizzati in un blocco (a differenza degli algoritmi a flusso, che operano su un singolo elemento alla volta)

[2] Algoritmo a chiave simmetrica: algoritmo che utilizza una una sola chiave sia per la fase di cifratura che per quella di decifratura dell'informazione. Quindi solo il mittente ed il destinatario devono conoscere entrambi la stessa chiave per potersi scambiare informazioni cifrate, per questo tale metodo è detto anche *cifratura a chiave segreta*.

il vantaggio di utilizzare poca memoria (il che lo rende facilmente installabile anche su smartcard) e inoltre mantiene le prestazioni anche al variare delle dimensioni della chiave e al variare delle piattaforme su cui viene utilizzato.

Funzionamento dell'algoritmo AES:

In questo algoritmo la lunghezza dei blocchi di cifratura è fissa ed è di 128 bit mentre le chiavi di cifratura possono essere di 128, 192 o 256 bit. L'algoritmo, per poter operare, ha bisogno di strutturare il testo in chiaro in matrici/blocchi 4x4 (4x4 bytes = 16 bytes = 128 bit) che vengono chiamati "states". Una volta strutturato il testo in chiaro, l'algoritmo AES suddivide la fase di cifratura ("round") in fasi identiche: ogni fase ha una sua sottochiave (o chiave di sessione) che viene estratta dalla chiave principale. Il numero delle fasi varia al variare delle dimensioni della chiave utilizzata. Ogni fase è costituita da quattro metodi, che applicati agli "states", attivano una serie di operazioni che lavorano per ottenere il testo cifrato:

- SubBytes: tutti i byte della matrice vengono sostituiti seguendo una specifica tabella in maniera non lineare;
- ShiftRows: si tratta di un cambio di posizione degli elementi di una riga. La prima riga della tabella rimane invariata, dalla seconda alla quarta viene sempre eseguito uno scorrimento circolare a sinistra di uno, due o tre bytes rispettivamente;
- MixColumns: i byte, una colonna per volta vengono trattati con un'operazione lineare (nello specifico si tratta di una moltiplicazione tra la matrice in questione e un polinomio fissato);
- AddRoundKey: viene eseguita una somma bit a bit modulo 2 tra la chiave segreta e lo "state". Ad ogni round la chiave aggiunta è diversa e ricavata dalle precedenti ricorsivamente.

```java
public class AESEncrypt implements EncryptStrategy{

    private String OutString;
    private String encryptString;

    public AESEncrypt() {}

    @Override
    public String encryptExecute(String txtFileCrypt, String key) {

        this.OutString = txtFileCrypt;
            try {
                byte[] byteMessage = this.OutString.getBytes();
                byte[] byteKey = key.getBytes();

                Key secretKey = new SecretKeySpec(byteKey, "AES") ;

                Cipher c = Cipher.getInstance("AES");
                c.init(Cipher.ENCRYPT_MODE, secretKey);
                byte[] cipher = c.doFinal(byteMessage);


                encryptString = Base64.getEncoder().encodeToString(cipher);

            } catch (Exception e) {
                e.printStackTrace();
            }
        return encryptString;
    }
}
```

Figura 18 - Classe AESEncrypt

5. AESDecrypt

La classe AESDecrypt serve a decodificare un testo attraverso l'algoritmo AES. In particolare, questo algoritmo non utilizza la stessa funzione per cifrare e decifrare ma due differenti. Il funzionamento della funzione per decifrare è molto simile al funzionamento della funzione per cifrare: ogni metodo applicato nelle fasi di cifratura ha un suo inverso che viene utilizzato nelle fasi di decifratura, tranne AddRoundKey che rimane invariato.

L'ordine di applicazione dei metodi è differente: viene eseguito il metodo InvShiftRows (inverso del metodo ShiftRows sopra descritto), poi il metodo InvSubBytes, il metodo AddRoundKey ed infine il metodo InvMixColumns.

```
public class AESDecrypt implements DecryptStrategy{

    private String OutString;
    private String decryptString;

    public AESDecrypt() {}

    @Override
    public String decryptExecute(String txtFileCrypt, String key) {

        this.OutString = txtFileCrypt;

        try {
            byte[] byteKey = key.getBytes();

            Key keyDec = new SecretKeySpec(byteKey, "AES");

            Cipher c = Cipher.getInstance("AES");
            c.init(Cipher.DECRYPT_MODE, keyDec);

            byte[] decodedValue = java.util.Base64.getMimeDecoder().decode(this.OutString);
            byte[] decValue = c.doFinal(decodedValue);

            decryptString = new String(decValue);

        } catch (Exception e) {
            System.out.println("Trouble reading from the file: " + e.getMessage());
        }
        return decryptString;
    }

}
```

Figura 19 – Classe AESDecrypt

6. CdCEncrypt

La classe CdCEncrypt serve per cifrare un testo con il metodo di cifratura Cifrario di Cesare. Il cifrario di Cesare funziona secondo il principio della sostituzione monoalfabetica, questa tecnica genera il testo cifrato sostituendo ogni lettera del testo in chiaro con la lettera che si trova un certo numero di posizioni dopo nell'alfabeto. La quantità di posizioni in cui spostare la lettera è gestita da una chiave numerica, Giulio Cesare per cifrare i suoi messaggi usava la chiave 3 (e.g. con questo metodo, utilizzando la chiave 3, la “A” in chiaro diventava la “D” nel testo cifrato, la “B” veniva sostituita con la “E” e così per tutte le lettere dell’alfabeto).
Il cifrario di cesare è ottimo da utilizzare per la sua semplicità ma purtroppo permette di cifrare messaggi aventi caratteri appartenenti solo all’alfabeto. Inoltre, gli spazi tra le parole di un messaggio vengono persi durante la cifratura del messaggio.

```java
public class CdCEncrypt implements EncryptStrategy{

    private String codString;

    public CdCEncrypt() {
        codString = "";
    }

    @Override
    public String encryptExecute(String txtFileCrypt, String key) {
        int shift = Integer.parseInt(key);
        txtFileCrypt = txtFileCrypt.replaceAll(" ", "");
        txtFileCrypt = txtFileCrypt.toUpperCase();

        for(int i = 0; i < txtFileCrypt.length(); i++) {
            if(txtFileCrypt.charAt(i) < 'A' || txtFileCrypt.charAt(i) > 'Z') {
                txtFileCrypt = txtFileCrypt.replaceAll(""+txtFileCrypt.charAt(i), "");
            }
        }

        codString = codifica(txtFileCrypt,shift);

        return codString;
    }

    private String codifica(String txtFileCrypt, int shift) {
        int val;
        String encryptString = "";

        for(int i = 0; i < txtFileCrypt.length(); i++) {
            val = (int)txtFileCrypt.charAt(i) + shift;
            if(val > 90)
                val -= 26;
            encryptString += (char)val;
        }

        return encryptString;

    }

}
```

Figura 20 - Classe CdCEncrypt

7. CdCDecrypt

La classe CdCDecrypt serve per decifrare un testo con cifrato con il Cifrario di Cesare.

```java
public class CdCDecrypt implements DecryptStrategy{

    @Override
    public String decryptExecute(String txtFileCrypt, String key) {

        int shift = Integer.parseInt(key);
        int val;
        String decryptString = "";

        for(int i = 0; i < txtFileCrypt.length(); i++) {
            val = (int)txtFileCrypt.charAt(i) - shift;
            if(val < 65)
                val += 26;
            decryptString += (char)val;
        }
        return decryptString;
    }
}
```

Figura 21 - Classe CdCDecrypt

8. FileLoader

La classe FileLoader è la classe che, incaricata dal pattern Virtual Proxy, carica il file di testo selezionato dall'utente, in modo da dare all'utente la possibilità di decifrarlo o, eventualmente, di modificarlo. Inoltre, questa classe è anche incaricata di caricare il file di log, visibile solo all'admin, CryptoFileLog.txt.

```java
public String openFile(String fileName) {

    try {
        FileReader fileReader = new FileReader(fileName);

        BufferedReader reader = new BufferedReader(fileReader);
        String tmpString = reader.readLine();
        int flag = 0;
        if(tmpString.equals("<%CRYPTED%>")) {
            flag = 1;
        }
        String OutString= tmpString.replace("<%CRYPTED%>", reader.readLine());
        if(flag == 1)
            isCrypted();

        tmpString = reader.readLine();

        while(tmpString != null) {
            OutString = OutString.concat("\n");
            OutString = OutString.concat(tmpString);
            tmpString = reader.readLine();
        }

        resultText = OutString;

        reader.close();
        fileReader.close();

    } catch (IOException ex) {
        System.out.println("Trouble reading from the file: " + ex.getMessage());
    }
    return resultText;
}

public boolean isCrypted() {
    return true;
}
```

Figura 22 - Metodo openFile() classe FileLoader

9. SaveFile

La classe SaveFile serve per andare a salvare il testo, creato o modificato dall'utente, sul terminale. In particolare, i due metodi principali di questa classe sono: writeOnFile e il metodo createNewFile. Il primo da la possibilità di sovrascrivere il testo caricato nello stesso file di testo mentre il secondo da la possibilità di creare un nuovo file contenente il testo creato dall'utente.

Questa classe svolge un ruolo molto importante in quanto estende la classe Observable, notificando all'observer che il soggetto è cambiato. In questo modo, ogni volta che l'utente andrà a salvare un file, il file di log verrà aggiornato.

Inoltre, per far capire all'applicazione se il file è stato cifrato o decifrato, al momento del salvataggio viene aggiunta, al testo da salvare, una stringa di caratteri (<%CRYPTED%>). In questo modo Data Lock sarà in grado di riconoscere se il file salvato era già cifrato oppure no. Se il file viene decifrato dall'applicazione, la stringa <%CRYPTED%> non verrà aggiunta al testo che verrà salvato.

Quando però il messaggio verrà mostrato all'utente, la stringa <%CRYPTED%> non verrà mostrata, nel caso il testo caricato fosse cifrato.

```
public String writeOnFile(String saveText) throws Exception {
    System.out.println(saveText);
    PrintWriter scriviPrintWriter = new PrintWriter(fileToSave);
    scriviPrintWriter.println("<%CRYPTED%>\n" + saveText);
    scriviPrintWriter.close();
    setChanged();

    return fileToSave.getName();
}
```

Figura 23 - Metodo writeOnFile()

```
public String createNewFile(String saveText) throws FileNotFoundException, IOException {

    File txtFile = new File("CryptedFile.txt");

    while(txtFile.exists()) {
        setNumber((int) Math.floor(Math.random() * 1000));
        txtFile = new File("CryptedFile"+getNumberFile()+".txt");
    }

    PrintWriter scriviPrintWriter = new PrintWriter(txtFile);
    scriviPrintWriter.println("<%CRYPTED%>\n" + saveText);
    scriviPrintWriter.close();
    setChanged();

    return txtFile.getName();
}
```

Figura 24 - Metodo createNewFile()

Design Patterns

Per l'implementazione dell'applicazione sono stati utilizzati alcuni design patterns. In particolare, i design patterns utilizzati sono:

- Observer
- Strategy
- Proxy
  - Protection Proxy
  - Virtual Proxy
- MVC

1. Observer

L'Observer è un design pattern di tipo comportamentale che si occupa di permettere a più istanze di una classe, detta Observer, di monitorare i cambiamenti di uno stato di un'istanza di un'altra classe, detta Subject, in modo automatico. Come accennato precedentemente, nel progetto le classi protagoniste di tale pattern sono SaveFile che svolge il ruolo di Subject e la classe LogSection che svolge il ruolo di Observer. L'obiettivo di tale struttura era creare un file di testo contenente un registro dei processi effettuati da DataLock, e dare la possibilità all'admin di leggerlo.

Ogni volta che un utente, base o admin, salva un file, viene invocato il metodo *setChanged()* che notifica all'Observer che lo stato attuale del Subject è cambiato. Una volta finito il processo di salvataggio del file, l'Observer invocherà il metodo *update()* per andare ad aggiornare il file di log.

```
@Override
public void update(Observable o, Object arg) { updateFileLog(); }
```

Figura 25 - Override metodo update()

2. Strategy

Il pattern Strategy è un pattern comportamentale utilizzato per avere un comportamento diverso di un metodo(incapsulto in un oggetto) a seconda della strategia scelta (in questo caso mediante i metodi *encryptExecute()* e *decryptExecute(),* poiché le strategie sono due). Nel progetto sono stati adottati due strategy pattern invece che uno, per tenere ben separate le strategie di cifratura e di decifratura. In particolare, attraverso il metodo *choosedStrategy()*, che interagisce con l'interfaccia grafica, si va a settare una delle due strategie disponibili andando così a cifrare o a decifrare il messaggio inserito dall'utente.

```
public String choosedStrategy(int typeOfStrategy, String MessageAreaString, String key) throws Exception{
    System.out.println("Message: " + MessageAreaString + " Type of Strategy: " + typeOfStrategy +" Key: " + key);
    switch (typeOfStrategy) {
        case 0: {
            return encrypt.setEncryptStrategy(algorithm, MessageAreaString, key);

        }case 1: {
            return decrypt.setDecryptStrategy(algorithm, MessageAreaString, key);
        }
        default: return null;
    }
}
```

Figura 26 - metodo choosedStrategy()

3. Protection Proxy

Il Protection Proxy, una versione del Proxy Design Pattern, è un pattern di tipo strutturale che si occupa di fornire un controllo sull'accesso a un'istanza di una classe remota. All'interno del progetto è stato utilizzato questo pattern per creare due livelli di accesso differenti. Questi due accessi vengono tutelati dall'immissione di campi obbligatori quali username e password per verificare l'identità dell'utente. La classe protagonista di tale struttura è LoginManagerProxy che estende l'interfaccia LoginInterface e monitora gli accessi alla classe IdentityManager.

```
public class LoginManagerProxy implements LoginInterface{

    private String username;
    private String password;
    private LoginManager loginManager;
    public int check;


    public LoginManagerProxy(String user, String password) {
        this.username = user;
        this.password = password;
        loginManager = new LoginManager();
    }

    @Override
    public int getAdvancedUser() {
        check = checkUser();
        return check;
    }

    @Override
    public List AccessList() {
        List lista = null;
        if( checkUser() == 0)
            lista = loginManager.AccessList();
        return lista;
    }

    private int checkUser() {
        return IdentityManager.checkUser(this.username, this.password);
    }
}
```

Figura 27 - Classe LoginManagerProxy

4. Virtual Proxy

Il Virtual Proxy, un'altra versione del Proxy Design Pattern, è un pattern di tipo strutturale che si occupa di ridurre il consumo delle risorse solo nel caso di utilizzo. L'obiettivo è di caricare direttamente il testo del documento quando viene richiesto. Questo consente di gestire in maniera oculata il caricamento di file critici. La classe protagonista è FileManagerProxy che estende l'interfaccia FileInterface che, nel momento necessario, istanzia la classe FileManager che a sua volta istanzierà la classe FileLoader che andrà a caricare il testo del file scelto.

```
public class FileManagerProxy implements FileInterface{

    private FileManager fileManager;
    private String filePath;

    public FileManagerProxy(String filePath) {
        this.filePath = filePath;
    }

    @Override
    public String displayFile() {
        if (fileManager==null)
            fileManager = new FileManager(filePath);

        return fileManager.displayFile();
    }

    public FileManager getFileManager() {
        return fileManager;
    }
}
```

Figura 28 - Classe FileManagerProxy

```
public class FileManager implements FileInterface{

    private String fileName;
    private FileLoader fileLoader;

    public FileManager(String fileName) {
        this.fileName = fileName;
        fileLoader = new FileLoader();
    }

    @Override
    public String displayFile() {
        return fileLoader.openFile(fileName);
    }

    public FileLoader getFileLoader() {
        return fileLoader;
    }
}
```

Figura 29 - Classe FileManager

```java
public class FileLoader {

    private String resultText = "";

    public FileLoader() {}

    public String openFile(String fileName) {

        try {
            FileReader fileReader = new FileReader(fileName);

            BufferedReader reader = new BufferedReader(fileReader);
            String tmpString = reader.readLine();
            int flag = 0;
            if(tmpString.equals("<%CRYPTED%>")) {
                flag = 1;
            }
            String OutString= tmpString.replace("<%CRYPTED%>", reader.readLine());
            if(flag == 1)
                isCrypted();

            tmpString = reader.readLine();

            while(tmpString != null) {
                OutString = OutString.concat("\n");
                OutString = OutString.concat(tmpString);
                tmpString = reader.readLine();
            }

            resultText = OutString;

            reader.close();
            fileReader.close();

        } catch (IOException ex) {
            System.out.println("Trouble reading from the file: " + ex.getMessage());
        }
        return resultText;
    }

    public boolean isCrypted() {
        return true;
    }
}
```

Figura 30 - Classe FileLoader

5. MVC

Il pattern architetturale Model-View-Controller ha lo scopo di separare la logica di dominio e di business di un programma dalla logica di presentazione.
Nell'applicativo in questione l'intero codice è stato suddiviso in 3 packages principali:

1. **Model**: il modello fornisce i metodi per accedere ai dati dell'applicazione. In questo caso il model coincide con le principali classi del progetto contenute nel package **model**.

2. **Controller**: svolge il ruolo da mediatore tra model e view. Riceve i comandi dall'utente e li attua modificando lo stato degli altri due componenti.
   In questo caso il controller coincide con la classe **Controller** nel package **controller**.

3. **View**: ha il compito di visualizzare i dati del model e di interagire con l'utente. In questo caso tutte le classi riguardanti l'interfaccia grafica utente (GUI) sono contenute nel package **view**.

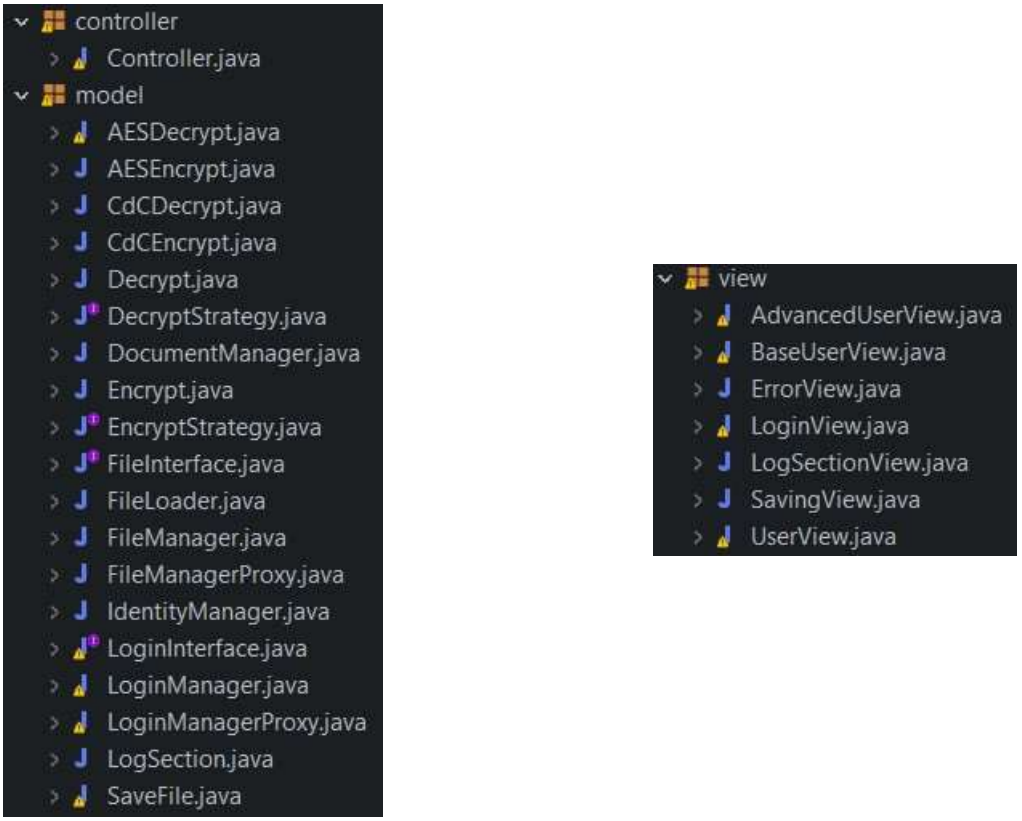


Figura 31 – Organizzazione package MVC

Ulteriori dettagli implementativi

- Per quanto riguarda la parte grafica, come già anticipato, è stato utilizzato il framework Java Swing con cui sono state realizzate le classi nel package **view**, estendendo la classe base JFrame.
- Per quanto riguarda la cifratura dei dati sono stati utilizzati il package javax.crypto e il package java.security. Inoltre, è stata adoperata la classe Base64 del package java.util e la classe java.crypto.spec.SecretKeySpec, che implementa l'interfaccia Key.
  La classe Cipher invece, fornisce le funzionalità di un algoritmo di cifratura. Di questa classe, come visto sopra nella figura 18 e nella figura 19, vengono utilizzati i seguenti metodi:
  - *getInstance()*: restituisce un oggetto Cipher che implementa la trasformazione specificata (AES);
  - *init()*: inizializza il cifrario con una chiave specifica (ENCRYPT_MODE è una costante intera che mette il cifrario in modalità "encrypt", allo stesso modo DECRYPT_MODE setta il cifrario in modalità "decrypt");
  - *doFinal()*: consente di cifrare o decifrare un messaggio.

Unit Testing

Quando si parla di Unit Test, si intende un test del software in cui vengono testate singole unità / componenti di un software. Lo scopo è convalidare che ogni unità del software funzioni come progettato. Un'unità è la parte testabile più piccola di qualsiasi software. Di solito ha uno o pochi input e di solito un singolo output. Nella programmazione procedurale, un'unità può essere un singolo programma, funzione, procedura, ecc. Nella programmazione orientata agli oggetti, l'unità più piccola è un metodo, che può appartenere a una classe base / super, una classe astratta o una classe derivata / figlia.

Nel progetto è stato utilizzato il framework **JUnit** nella versione 5.0.
In Junit i test-case sono anteposti dell'annotazione **@Test** (o eventualmente **@BeforeEach** oppure **@After**)
Per ogni classe principale del progetto sono state create le rispettive classi di Test contenenti i test-case relativi ai metodi della classe principale da testare.
Le classi di Test realizzate nel progetto sono:

- DocumentManagerTest
- DecryptTest
- EncryptTest
- FileLoaderTest
- IdentityTest
- LogSectionTest

1. DocumentManagerTest

In questa classe sono stati testati i metodi principali della classe DocumentManager, quali:

- *choosedStrategy()*
- *generateKey()*
- *saveFile()*
- *openFile()*

I test sono stati effettuati simulando le azioni che l'utente può andare ad eseguire, andando così a ricoprire tutte le casistiche possibili di interazione di questo con l'applicazione. Infatti, alcuni test (e.g. in *testSaveFile(),* figura 32) sono stati ripetuti con lo scopo di testare ogni singola possibilità di uso dell'applicazione.

```java
class DocumentManagerTest {

    DocumentManager docManager;
    SaveFile sFile;

    @BeforeEach
    void setUp() {
        docManager = new DocumentManager();
        sFile = new SaveFile();
    }

    @Test
    void testChooseStrategy() throws Exception {
        docManager.setAlgorithm(0);

        assertEquals(docManager.choosedStrategy(0, "Messaggio da Criptare", "VavElWbHQPasREDS"), "zcI8D/tgzzOMtW1n/Xxd7KRY/K8Qndb/FHAsbmh20Qk=");
        assertEquals(docManager.choosedStrategy(1, "zcI8D/tgzzOMtW1n/Xxd7KRY/K8Qndb/FHAsbmh20Qk=", "VavElWbHQPasREDS"), "Messaggio da Criptare");

        docManager.setAlgorithm(1);

        assertEquals(docManager.choosedStrategy(0, "Messaggio da Criptare", "3"), "PHVVDJJLRGDFULSWDUH");
        assertEquals(docManager.choosedStrategy(1, "PHVVDJJLRGDFULSWDUH", "3"), "MESSAGGIODACRIPTARE");
    }

    @Test
    void testGenerateKey() throws NoSuchAlgorithmException {
        docManager.setAlgorithm(0);
        assertEquals(docManager.generateKey().length(), 16);
        docManager.setAlgorithm(1);
        assertTrue(Integer.parseInt(docManager.generateKey()) < 26);
    }

    @Test
    void testSaveFile() throws Exception {


        sFile.setAdmin(false);

        docManager.setFileLoaded(true);
        docManager.setFileToSave(new File("CryptedFile.txt"));
        docManager.saveFile("Stringa da Salvare");

        assertEquals(docManager.isOverwritten(), true);

        docManager.setFileLoaded(false);
        docManager.setFileToSave(null);
        docManager.saveFile("Stringa da Salvare");
        assertEquals(docManager.isOverwritten(), false);
    }

    @Test
    void testOpenFile() {
        assertTrue(docManager.openFile("C:/Users/piero/Desktop/CryptoFileLog.txt"), true);

        docManager.setIsCrypted(docManager.getFileManagerProxy().getFileManager().getFileLoader().isCrypted());
        assertFalse(docManager.getIsCrypted());
    }
}
```

Figura 31 - Classe DocumentManagerTest

2. DecryptTest

In questa classe è stato testato il settaggio della strategia Decrypt, in modo da andare a verificare che tutti i dati che vengono passati alla strategia vengano elaborati restituendo così dei valori corretti.

Sono quindi state testate le possibilità di settaggio della strategia Decrypt, sia con lo algoritmo AES sia con l'utilizzo del cifrario di Cesare.

```java
class DecryptTest {

    Decrypt decrypt;

    @BeforeEach
    void setUp() {
        decrypt = new Decrypt();
    }

    @Test
    void testSetDecryptStrategy() {
        assertEquals(decrypt.setDecryptStrategy(0, "zcI8D/tgzzOMtW1n/Xxd7KRY/K8Qndb/FHAsbmh20Qk=", "VavElWbHQPasREDS"), "Messaggio da Criptare");
        assertEquals(decrypt.setDecryptStrategy(1, "PHVVDJJLRGDFULSWDUH", "3"), "MESSAGGIODACRIPTARE");
    }
}
```

Figura 32 - Classe DecryptTest

3. EncryptTest

Come nella classe precedente, DecryptTest, è stato testato il metodo che setta la strategia Encrypt in base alla scelta dell'utente. Anche in questo caso sono stati testati sia l'algoritmo AES che il cifrario di Cesare.

```
class EncryptTest {

    Encrypt encrypt;

    @BeforeEach
    void setUp() {
        encrypt = new Encrypt();
    }
    @Test
    void testEncryptStrategy() {
        assertEquals(encrypt.setEncryptStrategy(0, "Messaggio da Criptare" , "VavElWbHQPasREDS"), "zcI8D/tgzzOMtW1n/Xxd7KRY/K8Qndb/FHAsbmh20Qk=");
        assertEquals(encrypt.setEncryptStrategy(1, "Messaggio da Criptare" , "3"), "PHVVDJJLRGDFULSWDUH");
    }
}
```

Figura 33 - EncryptTest

4. FileLoaderTest

In questa classe è stato testato l'utilizzo del pattern Virtual Proxy, forzando l'apertura di un file specifico, cifrato, salvato sul terminale. È stato testato che il testo caricato dal file combaci con quello effettivamente presente sul file ma senza la stringa di identificazione del file cifrato. Inoltre, è stato testato anche il passaggio dell'attributo isCrypted che permette all'applicazione di sapere se il file caricato era cifrato.

```
class FileLoaderTest {

    DocumentManager dManager;
    FileManagerProxy fProxy;

    @BeforeEach
    void setUp() throws Exception {
        dManager = new DocumentManager();
        fProxy = new FileManagerProxy("fileTest.txt");
    }

    @Test
    void testIsCrypted() {
        assertEquals(fProxy.displayFile(), "6JOf2PdcX/5zroHLZWODUuNwNlpLBJOXiZdACL3hqt0BCrogFr3vV72svBzYuoM+");
        assertEquals(fProxy.getFileManager().getFileLoader().isCrypted(), true);
    }
}
```

Figura 34 - Classe FileLoaderTest

5. IdentityTest

In IdentityTest, è stato testato l'utilizzo del pattern Protection Proxy. Per testare il pattern è stato creato a mano un LoginManagerProxy, con parametri user e password precisi, e successivamente è stato controllato l'accesso alla LogSection, andando a controllare che l'attributo isAdmin fosse settato in maniera corretta in base all'utente che tenta l'accesso.

```java
class IdentityTest {

    LoginInterface lInterface;
    LogSection ls;

    @BeforeEach
    void setUp() {
        ls = new LogSection();
    }

    @Test
    void test() {
        lInterface = new LoginManagerProxy("root", "password");

        assertEquals(lInterface.getAdvancedUser(), 2);
        assertFalse(ls.getIsAdmin());

        lInterface = new LoginManagerProxy("root", "toor");

        assertEquals(lInterface.getAdvancedUser(), 0);
        assertTrue(ls.getIsAdmin());
    }
}
```

Figura 35 - Classe IdentityTest

6. LogSectionTest

In questa classe è stato realizzato un test per verificare il corretto aggiornamento del file di log, ovvero è stato testato il corretto funzionamento del pattern Observer. Per realizzare questo test, è stato misurato la dimensione del file di log prima e dopo il salvataggio di un nuovo file. Alla fine, queste due dimensioni (in byte) sono state comparate tra di loro, aspettandosi che la seconda misurazione, della dimensione del file di log, fosse maggiore rispetto alla prima misurazione fatta.

```java
class LogSectionTest {

    Encrypt encrypt;
    DocumentManager dManager;

    @BeforeEach
    void setUp() throws Exception {
        encrypt = new Encrypt();
        dManager = new DocumentManager();
    }

    @Test
    void test() throws Exception {

        File fileToCheckBefore = new File("C:/Users/piero/Desktop/CryptoFileLog.txt");
        long sizeInBytesBefore = fileToCheckBefore.length();

        dManager.setFileLoaded(false);
        dManager.saveFile("Salvo una stringa per il test");

        File fileToCheckAfter = new File("C:/Users/piero/Desktop/CryptoFileLog.txt");
        long sizeInBytesAfter = fileToCheckAfter.length();

        assertTrue(sizeInBytesAfter > sizeInBytesBefore);
    }

}
```

Figura 36 - Classe LogSectionTest