KeePassJ per "Programmazione ad Oggetti"

Di Santi Giovanni, Ercolani Francesco, Conti Massimiliano, Cellot Davide

27 settembre 2020

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di realizzare un keepass per salvare in modo sicuro le password. Al momento i keepass desktop più usati sono keepass2 [**?**] e keepassxc [**?**]. TODO: scrivere descrizione più lunga.

Requisiti funzionali

- Il database su cui sono salvate le password deve essere opportunamente autenticato e cifrato, in modo da non permettere a terze parti di leggere o manipolare il contenuto delle password.
- Gestione delle entry (add, edit, delete) con la possibilità di suddividere le entry in vari gruppi.
- Funzioni per generare in modo sicuro password e controllarne la validità.
- Possibilità di importare in chiaro un database in formato XML, in modo simile alla funzionalità di un keepass originale.
- Sezione Statistics che mostra le statistiche relative al proprio database (Es. il numero di account salvati)

Requisiti non funzionali

- Possibilità di cifrare archivi esterni.
- Funzioni di sort e find per ordinare e cercare entry specifiche.

- Funzione expire che avvisa quando una password è da cambiare (es. 2 anni).

1.2 Analisi e modello del dominio

Nella creazione del database bisogna inserire una master password. Dopodiché è necessario configurare vari parametri, tra cui il cipher e la Key Derivation Function (KDF) da usare. La KDF ha vari parametri opzionali da utilizzare per generare la key in modo più sicuro (più rounds) o più veloce (più parallelismo). Lo pseudocodice seguente spiega come la **KDF** e il **cipher** sono collegati:

```
plaintext = "this is the plaintext"
key = KDF(password)
ciphertext = cipher.encrypt(key, plaintext)
assert plaintext == cipher.decrypt(key, ciphertext)
```

I vari parametri crittografici del database sono configurabili tramite **KDBHeader**, che vengono usati dalla classe **KDB** per effettuare l'encryption e la decryption di array di byte arbitrari. Per lavorare sul database in chiaro usiamo un ulteriore classe che è **DataBase** che permette di manipolare le entry e i gruppi.

Figura 1.1: rappresentazione UML dell'analisi del progetto

Capitolo 2

Design

2.1 Architettura

Per la realizzazione di KeePassJ abbiamo scelto di utilizzare il pattern architetturale Model-View-Controller (MVC).

TODO: Inserire schema UML MVC.

2.2 Design dettagliato

Giovanni Di Santi

Il mio compito principale del progetto è stato quello di gestire la parte crittografica e definire la struttura dell'header del database. Prima di iniziare a modellare le varie parti mi sono documentato sulle tecnologie attuali dei vari keepass, il paper più citato è "On The Security of Password Manager Database Formats" [**?**]. Mi sono ispirato a come funziona il formato di keepass2 KDBX4. Il formato è composta da un *header* e un *body*. Nell'*header* sono definiti i vari cipher, kdf, e parametri per cifrare/decifrare il *body*. Il body decifrato in KDBX4 è formato da un XML, tuttavia questa parte è ha scelta di chi deve implementare il database in chiaro. Nel nostro caso Massimiliano ha deciso di usare XML perché ha una buona interoperabilità con Java.

La sezione del paper *4.6* evidenzia come il formato KDBX4 di keepass2 sia insicuro contro attacchi del tipo **MAL-CDBA**. Senza entrare nei dettagli, l'header del database non viene autenticato e quindi si aprono una serie di possibili attacchi teorici. Tuttavia il paper sopra citato non è coerente con il sito di keepass2 in cui si descrive KDBX4, infatti leggendo il sorgente di keepass2 si può notare che l'autenticazione viene eseguita. Probabilmente il paper essendo del 2012 non è aggiornato sui recenti cambiamenti.

CryptoCipher

CryptoCipher è l'interfaccia che descrive i metodi necessari per effettuare l'encryption e la decryption di un array di `byte`.
Ogni implementazione disponibile di questa interfaccia è un AEAD Cipher [**?**] (Authenticated Encryption with Associated Data).
Ho scelto questo schema di encryption per rendere il database resistente ad attacchi del tipo **CCA** (Chosen Ciphertext Attack), cifrando il contenuto del database e autenticando sia il contenuto che l'header (Associated Data).
Attualmente i cipher disponibili sono:

Figura 2.1: rappresentazione UML del pattern factory per creare un CryptoCipher

- ChaCha20-Poly1305 [**?**].
- AES-GCM [**?**].
- AES-256-CBC-HMAC-SHA-512 [**?**].

Esistono due `abstract class` diverse per implementare un **CryptoCipher**, poiché la costruzione di **AES** che sarebbe `AES-256-CBC-HMAC-SHA-512` è manuale, mentre **ChaCha20-Poly1305** e **AES-GCM** sono implementate direttamente in openjdk11. La classe astratta **AEADAES**, permette di essere estesa per costruire altri cryptosystem come `AES-192-CBC-HMAC-SHA-384`, tuttavia ho deciso di estendere solo lo schema più sicuro.
Nonostante i dati da cifrare e decifrare sono nella pratica degli {Input,Output}Stream, non ho usato le classi CipherOutputStream e CipherInputStream per:

- Rendere più semplice il suo utilizzo.
- Facilitare il testing delle varie implementazioni.

KDF

KDF (Key Derivation Function) è l'interfaccia che descrive i metodi necessari e opzionali per generare una chiave simmetrica per cifrare/decifrare il database.
Gli algoritmi disponibili sono:

- Argon2.
- Scrypt.

Figura 2.2: rappresentazione UML del pattern factory per creare un KDF

- PBKDF2.

Argon2 e **Scrypt** estendono **KDFAdvanced** poiché i loro algoritmi permettono di definire parametri extra come il paralellismo e la memoria usata dal KDF. **PBKDF2** è un vecchio metodo per generare una chiave dalla password e l'unico parametro configurabile è il numero di round che usa internamente, per questo ho settato il campo `tweakable` a falso.

Per capire perché il pattern `Factory` è usato sia per creare **KDF** e **CryptoCipher** bisogna prima analizzare il parsing dell'header e la relativa encryption/decryption del database.

KDB

Per progettare questa parte non ho usato le interfacce perché:

- Ho solo una implementazione disponibile.
- Sono più flessibile quando devo cambiare la signature di un metodo, senza dover usare un IDE o un LSP per il refactoring.
- Principio YAGNI e KISS.

KDBHeader è la classe che si occupa di:

- Parsare l'header (Lettura).
- Configurare i vari parametri (Scrittura).

KDB è la classe che tramite il **KDBHeader** si occupa di cifrare/decifrare dati arbitrari. Il pattern factory per **CryptoCipher** e **KDF** è utile quando in **KDB** si effettua l'operazione `encrypt` e `decrypt`. I metodi richiedono a **KDBHeader** il valore (String) del `Cipher` e del `KDF` che viene passato come parametro di {Cipher,KDF}Factory.create() per generare l'oggetto richiesto. I due metodi pubblici principali di **KDB** sono:

- `write`: che esegue l'encryption dell'array di byte in input e lo scrive sul file. Ad ogni write viene generato un nuovo IV, che viene poi usato per cifrare il plaintext. In questo modo si creano due versioni diverse dello stesso *body* e si rende lo schema crittografico sicuro contro vari attacchi di tipo **CCA**.

- `read`: che legge il file e lo decrypta. Questo metodo lancia un `IOException` se il file non esiste o un `AEADBadTagException` quando il file è corrotto. Il file può risultare corrotto o perché la password è sbagliata o perché è stato effettivamente manipolato. Non lancio tipi diversi di eccezioni (es. `BadPaddingException`) in base a vari tipi di errore, per evitare vari tipi di attacchi (praticamente difficili, ma teoricamente possibili) come il padding oracle.

Figura 2.3: rappresentazione UML di KDB

Francesco Ercolani

Il mio ruolo nel progetto KeePassJ è stato quello di sviluppo e gestione dell'interfaccia grafica del programma. L'interfaccia grafica è gestita attraverso controllers situati nel package **view.controllers**, ciascuno collegato al rispettivo file fxml presente nella directory **src/main/resources/view**. I controllers utilizzano classi nel package controller per gestire la logica dell'interfaccia.

view.controllers

Come già citato in precendenza, il package **view.controllers** contiene tutti i controllers dei rispettivi file fxml; il mio compito è stato quello di creare sia i file fxml che di gestire la logica, con opportuni metodi e campi interni ai rispettivi controllers, di una parte di essi. La directory src/main/resources/ è gestita nel seguente modo: all'interno della sottodirectory **/view**, è presente il file **MainMenu.fxml** e due ulteriori sottodirectory **/createnew** e **/database**. All'interno di /createnew sono presenti i file:

- **chooseNameDb.fxml**: è l'interfaccia per l'inserimento del nome e della descrizione del database.
- **chooseEncryptionSet.fxml**: è l'interfaccia per l'impostazione del metodo di encryption del database.
- **choosePassMenu.fxml**: è l'interfaccia per la scelta della password del database.

All'interno di /database sono presenti i file:

- **OpenDatabase.fxml**: è l'interfaccia per l'inserimento della password per l'apertura del database.
- **ManageMenu.fxml**: è l'interfaccia dove vengono visualizzati gli account registrati nel database corrente e dove si può scegliere di aggiungere un account o un gruppo.
- **AddEntry.fxml**: è l'interfaccia dove si aggiungono gli account che si vogliono gestire.
- **AddGroup.fxml**: è l'interfaccia dove si aggiungo i gruppi ai quali apparterranno gli account inseriti.

Il file MainMenu.fxml che si trova dentro la directory /view è l'interfaccia principale che viene caricata all'esecuzione del programma e contiene i due pulsanti principali: "Create new database" e "Open database" per appunto rispettivamente creare ed aprire un database. Oltre che della creazione dei file fxml citati, mi sono occupato della creazione di tutti i rispettivi controllers e della gestione della logica interna di essi eccetto per: **AddEntryController.java**, **AddNewGroupController.java** e **ManageMenuController.java** (compito delegato ad altri componenti del gruppo di progetto).
Dal controller di MainMenu.fxml **MainMenuController.java** partono due "linee" separate in riferimento alla creazione o all'apertura di un database.
Nel caso si clicchi su "Create new database", la linea prosegue verso il controller dell'interfaccia per l'inserimento del nome e della descrizione del database (chooseNameDb.fxml) **ChooseNameDBController.java**. Dentro quest'ultimo sono di fondamentale importanza i campi per l'acquisizione dei dati inseriti dall'utente.
Cliccando su "Continue" la linea prosegue passando al controller dell'interfaccia adibita al settaggio dell'encryption del database **ChooseEncrSetController.java**, che a sua volta acquisirà anch'esso i dati inseriti dall'utente.
Continuando con la creazione si arriva all'ultima interfaccia della linea controllata dal controller **ChoosePassController.java**, il quale alla pressione del bottone "Confirm", attraverso l'oggetto di tipo KDBHeader, setta tutti i valori inseriti dall'utente.
La seconda linea invece porta all'interfaccia di inserimento della password per l'apertura del database selezionato in precedenza.

I controllers al loro interno hanno un metodo per ciascuna azione che l'utente compie nell'interfaccia.
Ciascun controller fa uso di classi presenti nel package **controller**. Per il

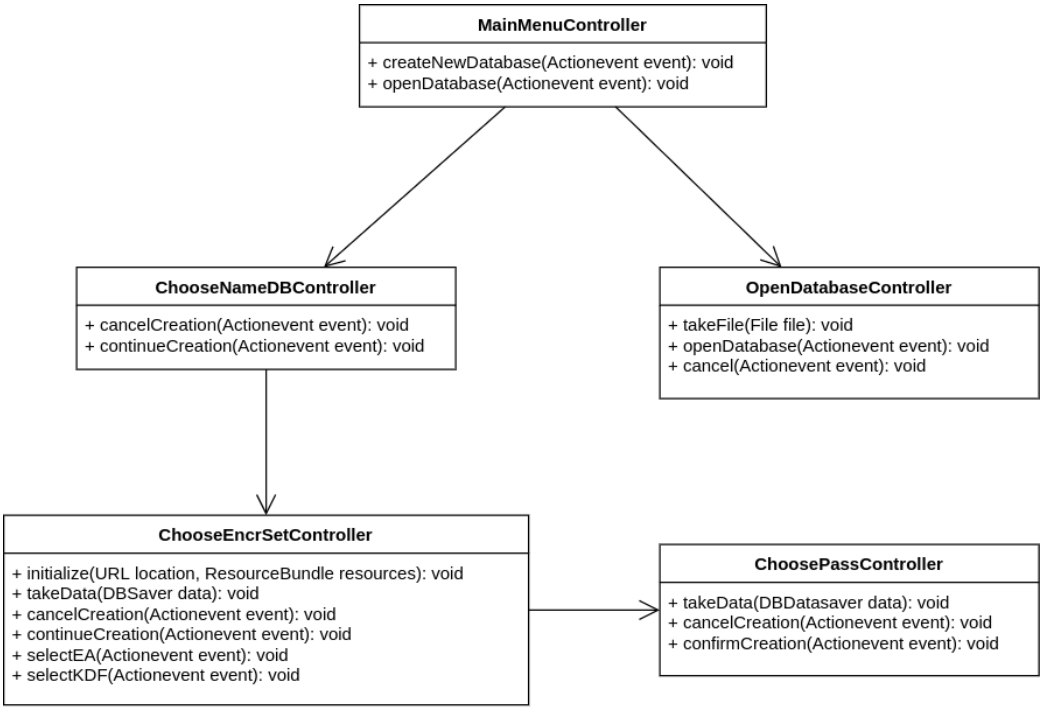

Figura 2.4: rappresentazione UML delle due linee che collegano i controllers

loading dei file fxml si utilizza la classe **FxmlFilesLoaderImpl.java** che implementa la relativa interfaccia **FxmlFilesLoader.java**.
Questa classe ha diversi metodi per fare il caricamento dell'interfaccia in modi differenti: alcuni caricano il file con metodo standard, altri hanno parametri per il passaggio di dati da controller a controller, infatti alcuni controller come da OpenDatabaseController.java in poi, hanno bisogno dei dati inseriti precedentemente.
Per implementare questo passaggio ho creato una classe **DBDataSaverImpl.java** che salva i dati inseriti e li passa, attraverso metodi di FxmlFilesLoaderImpl, al controller successivo.
È stato scelto questo approccio per due principali motivi:

- In questo modo la conferma e il settaggio dei dati avviene soltanto alla fine del procedimento di creazione.
- Il passaggio da controller a controller è più semplice in termini di trasferimento.

Ogni controller che richiede dati dal precedente ha quindi un metodo **takeData()** che prende in ingresso l'oggetto di tipo DBDataSaver. I con-

trollers utilizzano inoltre la classe **FxmlSetterImpl.java** che implementa l'interfaccia **FxmlSetter.java**, per operazioni come:

- Settaggio di spinner o warning dialogs.
- Acquisizione dello stage corrente.

Il settaggio effettivo per la creazione avviene in ChoosePassController.java, alla pressione del pulsante "Confirm" dopo aver scelto il nome e la destinazione del file (deve avere necessariamente l'estensione .kdbj), attraverso le classi KDBHeader e KDB.

Se si sceglie di aprire un database esistente, si apre la schermata di inserimento della password controllata da **OpenDatabaseController.java** e, in caso di inserimento corretto, si apre la schermata di gestione degli account controllata da **ManageMenuController.java**.
Anche in questo caso il passaggio di dati tra controllers avviene attraverso l'utilizzo della classe **FxmlFilesLoader**.

Vantaggi di questo approccio per il passaggio dei dati:

- Semplicità di utilizzo: è facile passare i dati da un controller all'altro in quanto viene passato soltanto l'oggetto responsabile del raggruppamento di essi.
- Non c'è bisogno di settaggi intermedi sugli oggetti KDBHeader e KDB in quanto viene settato tutto alla conferma finale.

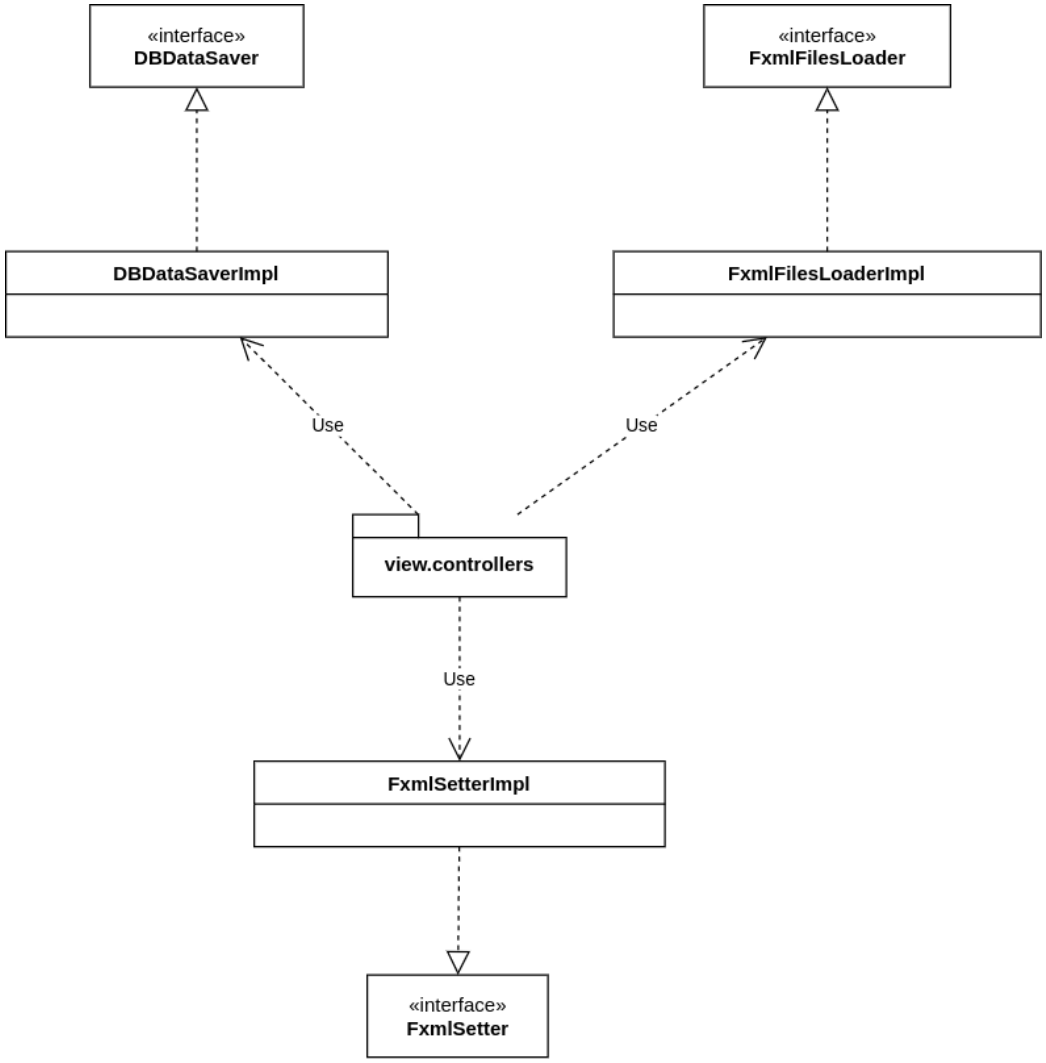

Figura 2.5: rappresentazione UML della relazione tra le classi nel package controller e i controllers in view

Svantaggi di questo approccio per il passaggio dei dati:

- Ridondanza di codice nella classe FxmlFilesLoaderImpl.
- Necessità di modifiche/aggiunte nella classe FxmlFilesLoaderImpl nel caso di cambi/aggiunte di funzionalità nell'applicazione.

Esecuzione dell'applicazione

La classe **ViewImpl.java** che implementa l'interfaccia **View.java** nel package view, è incaricata del loading della prima scena del programma. La classe **Main.java**, situata all'interno del package **application**, estende la classe Application di javafx e implementa quindi il metodo **start()** che è il main entry point dell'applicazione. All'interno del metodo main il programma viene lanciato con il metodo statico **launch()**.

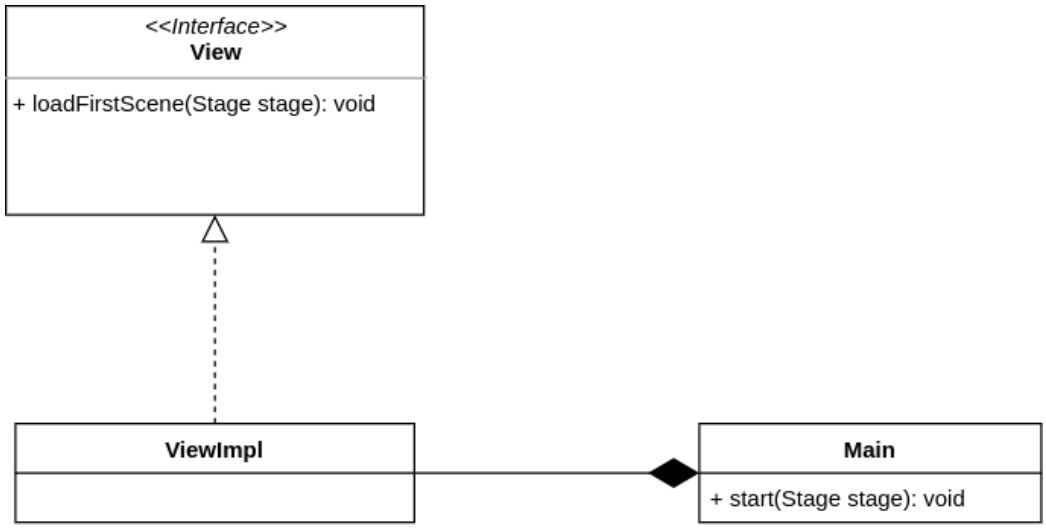

Figura 2.6: rappresentazione UML dell'esecuzione dell'applicazione

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Durante lo sviluppo del nostro progetto abbiamo utilizzato `Junit` per testare il corretto funzionamento delle varie classi. **Funzionalità testate automaticamente**:

- *CryptoCipher*: viene testato il corretto comportamento dei vari cipher utilizzando lo specifico `Factory`.
- *KDF*: vengono testate le varie KDF utilizzando lo specifico `Factory`.
- *Crypto Util*: test di varie utility per crittografia, come il PKCS#7 padding e SHA256.
- *KDBHeader*: viene testato il corretto funzionamento del **KDBHeader**, settando vari parametri e controllando il corretto funzionamento dei getter.
- *KDB*: test sulla scrittura e lettura di diverse combinazioni di `cipher` e `KDF`.

I test vengono eseguiti anche in remoto tramite l'apposita `bitbucket pipelines` che esegue la build del progetto e lancia i vari test.

3.2 Metodologia di lavoro

TODO.

3.3 Note di sviluppo

Giovanni Di Santi

- *javax.crypto e java.security*: usata per lavorare con Cipher, KDF, e MessageDigest.
- *Stream*: usata per manipolare l'header in modo efficace ed elegante.
- *Libreria Google Guava*: lavorare con i byte array in java non è comodo. Questa libreria ha varie classi per semplificare il lavoro tra cui Bytes.
- *ByteBuffer*: per convertire in little endian i bytes, i Data{Input,Output}Stream in java lavorano solo in big-endian.
- *Libreria Argon2 e Scrypt*: Questi due KDF non erano disponibili dentro `javax.crypto`.
- *Libreria Apache Commons*: per convertire byte array in formato esadecimale.

Ho iniziato lo sviluppo leggendo i sorgenti di keepass2, keepassxc e di pykeepass. Non ho trovato librerie simili a construct in java per parsare l'header utilizzando un linguaggio dichiarativo. L'unica alternativa era kaitai.io, ma il supporto a java era limitato. Avendo avuto esperienze con le librerie crittografiche in python e C, il passaggio a java non è stato difficile. Per i pattern progettuali ho consultato il materiale didattico e online, ho trovato che il pattern factory mi aiutasse a risolvere vari tipi di problemi per la generazione automatica di oggetti e l'ho usato.

Francesco ercolani

In quanto incaricato di sviluppare la maggior parte dell'interfaccia grafica dell'applicazione, ho fatto un largo uso della libreria esterna **JavaFX**, in particolare dei moduli:

- **javafx.controls**: per la gestione dei componenti grafici.
- **javafx.base**: per la gestione degli elementi logici (proprietà, collezioni, eventi ecc...).
- **javafx.fxml**: per la gestione dei file FXML.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Giovanni Di Santi

L'idea iniziale del progetto è stata mia perché mi piace lavorare con la crittografia. Mi è piaciuto come ho realizzato la parte di **CryptoCipher**, tuttavia l'architettura che riguarda **KDF** non mi sembra eccellente. Un difetto che ho trovato è stata l'integrazione manuale dei vari metodi in **KDF** dentro gli accessor method di **KDBHeader**. Dovrebbero esistere librerie, plugin o pattern di programmazione su misura per quel tipo di problema ma non li ho usati. In **KDB** non ho aderito al Single-responsibility principle (SRP) per comodità, infatti la classe effettua sia la `write` che la `read`, al posto di suddividere i compiti in due classi diverse. L'ho fatto principalmente perché entrambi i metodi usano lo stato dell'oggetto (campi) per essere eseguiti, quindi era molto conveniente tenerli all'interno della stessa classe.

Francesco Ercolani

Il mio ruolo all'interno del progetto KeePassJ è stato più difficile di quanto mi aspettassi. Le interfacce da me create sono molto basiche in quanto la mia esperienza con la librearia JavaFX era nulla, e per questo spero di trovare il tempo per proseguire lo sviluppo estetico su un branch personale. La gestione logica dei controller non è ottimale in quanto fa uso di classi da me create che contengono codice ridondante. L'applicazione in sè è abbastanza "povera" di grafica in quanto priva di diverse funzionalità presenti invece in password manager attualmente in commercio. Il progetto è molto valido, le applicazioni password manager sono estremamente utili e ricoprono una grande responsabilità nel mantenimento in sicurezza dei propri dati personali,

ed è per questo che ne sono stato entusiasta fin dall'inizio quando mi è stato proposto come idea da Giovanni Di Santi. Lo sviluppo di questo processo, nonostante i problemi riscontrati, mi ha fatto trovato la voglia di progettare un'applicazione da zero e, allo stesso tempo, come già detto, spero in futuro di ritagliarmi uno spazio per proseguire lo sviluppo di KeePassJ.

4.2 Difficoltà incontrate e commenti per i docenti

Giovanni Di Santi

Ho dovuto lavorare di più del dovuto perché i miei compagni fino a settembre non hanno implementato nessuna parte di codice significativo. A inizio giugno la modellazione e l'implementazione della maggior parte del mio compito erano completate, tuttavia mi sono ritrovato ad aggiungere parti di codice in più a settembre, poiché i miei compagni hanno iniziato a lavorare solo in quel mese. Vari problemi sarebbero nati subito se avessimo lavorato insieme. Ad esempio ho dovuto aggiungere vari metodi che aiutassero Francesco a modellare la GUI, difatti tutti i miei commit di settembre sono getter e setter aggiuntivi. Avendo progettato l'architettura abbastanza bene non ho avuto problemi ad aggiungere quelle parti di codice, tuttavia ho dovuto riprendere in mano un codice scritto 3 mesi prima e continuare a lavorarci sopra. Avrei preferito consegnare con la deadline C.

L'intenzione iniziale quando ho inziato a lavorare sul progetto era di renderlo compatibile con KDBX4, tuttavia ho notato dopo le prime 15 ore che quel formato è molto complicato da implementare. Non esiste documentazione del formato e quindi ho speso molto tempo a reverse ingegnerizzare il codice originale. Dopo le prime ore di implementazioni ho mantenuto la struttura, ma ho deciso di usare nuovi schemi crittografici. Non mi sono pentito della scelta fatta poiché mi ha reso più creativo.

Francesco Ercolani

L'architettura del progetto è rimasta molto vaga per i primi mesi dopo l'avvio del progetto. Inizialmente le funzionalità di KeePassJ dovevano essere tante e questo ha suscitato in me da subito perplessità riguardo al fatto di riuscire a sviluppare un software con così tante features complesse in poco tempo. Inoltre la situazione Covid, e quindi l'impossiblità di vedersi fisicamente, ha creato all' interno del gruppo diverse difficoltà di comunicazione dovute a problemi tecnici e/o personali. L'inizio dello sviluppo è partito in

momenti diversi da parte dei componenti e la mancanza di collaborazione e di specifiche su come si dovessero svolgere le parti assegnate ha creato ulteriori difficoltà di comprensione. Personalmente ritengo che fino al mese di agosto ci sia stata troppa poca collaborazione e troppo poco impegno nel delineare **chiaramente** quali fossero **concretamente** le parti software da sviluppare. Mi prendo le mie responsabilità per il ritardo di consegna della mia parte, ma ritengo che parte dei problemi siano nati da situazioni che non dipendono da me o dagli altri componenti del gruppo. Tuttavia nel complesso reputo che il lavoro svolto sia comunque molto buono data l'inesperienza progettuale e di linguaggio comune a tutti e 4.

Appendice A

Guida utente

Capitolo in cui si spiega come utilizzare il software. Nel caso in cui il suo uso sia del tutto banale, tale capitolo può essere omesso. A tal riguardo, si fa presente agli studenti che i docenti non hanno mai utilizzato il software prima, per cui aspetti che sembrano del tutto banali a chi ha sviluppato l'applicazione possono non esserlo per chi la usa per la prima volta. Se, ad esempio, per cominciare una partita con un videogioco è necessario premere la barra spaziatrice, o il tasto "P", è necessario che gli studenti lo segnalino.

Elementi positivi

- Si istruisce in modo semplice l'utente sull'uso dell'applicazione, eventualmente facendo uso di schermate e descrizioni.

Elementi negativi

- Si descrivono in modo eccessivamente minuzioso tutte le caratteristiche, anche minori, del software in oggetto.
- Manca una descrizione che consenta ad un utente qualunque di utilizzare almeno le funzionalità primarie dell'applicativo.