

KeePassJ per “Programmazione ad Oggetti”

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

25 settembre 2020

Indice

1	Analisi	2
1.1	Requisiti	2
1.2	Analisi e modello del dominio	3
2	Design	4
2.1	Architettura	4
2.2	Design dettagliato	4
3	Sviluppo	9
3.1	Testing automatizzato	9
3.2	Metodologia di lavoro	9
3.3	Note di sviluppo	10
4	Commenti finali	11
4.1	Autovalutazione e lavori futuri	11
A	Guida utente	13

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 **KDB-Header**, 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 permetterà di manipolare le entry e i gruppi.

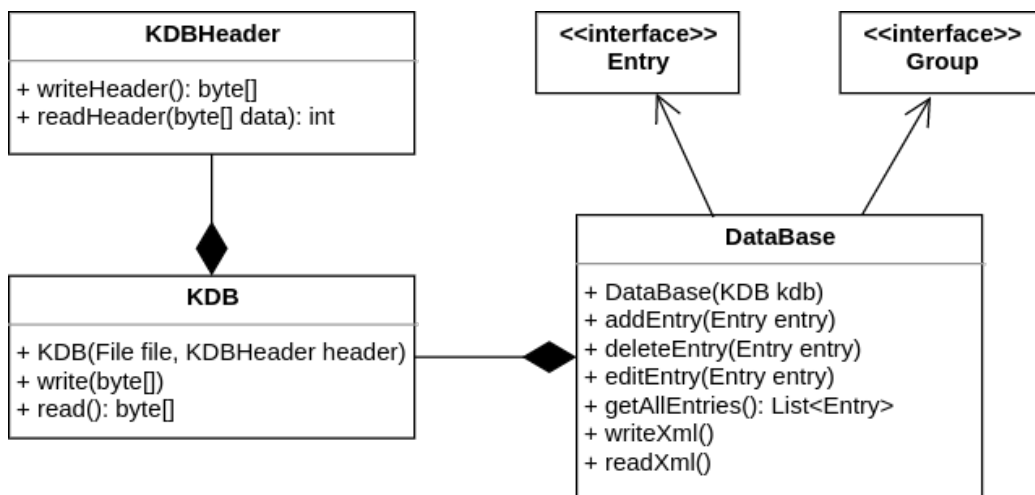


Figura 1.1: rappresentazione UML dell'analisi del progetto

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

TODO: Spiegare a grandi linee

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.

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 [4] (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:

- ChaCha20-Poly1305 [2].

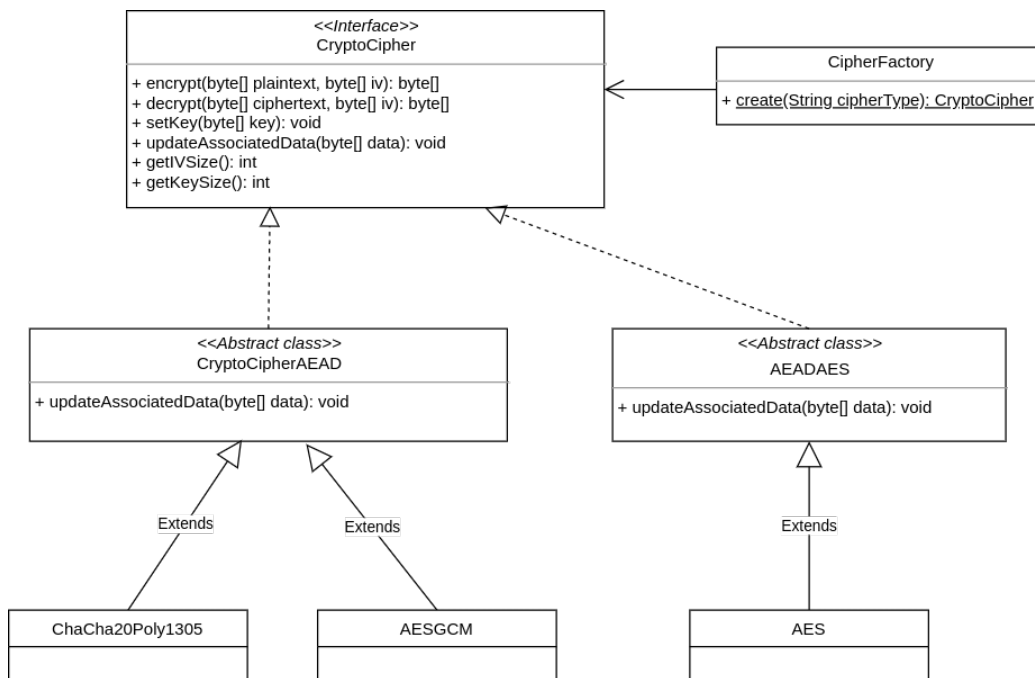


Figura 2.1: rappresentazione UML del pattern factory per creare un CryptoCipher

- AES-GCM [3].
- AES-256-CBC-HMAC-SHA-512 [1].

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.
- PBKDF2.

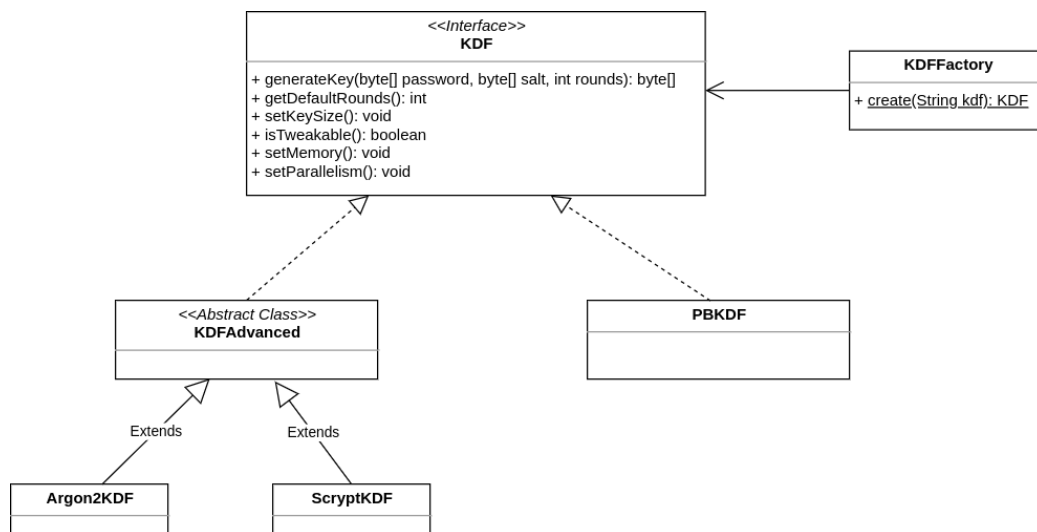


Figura 2.2: rappresentazione UML del pattern factory per creare un KDF

Argon2 e **Scrypt** estendono **KDFAdvanced** poiché i loro algoritmi permettono di definire parametri extra come il parallelismo 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.
- **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.

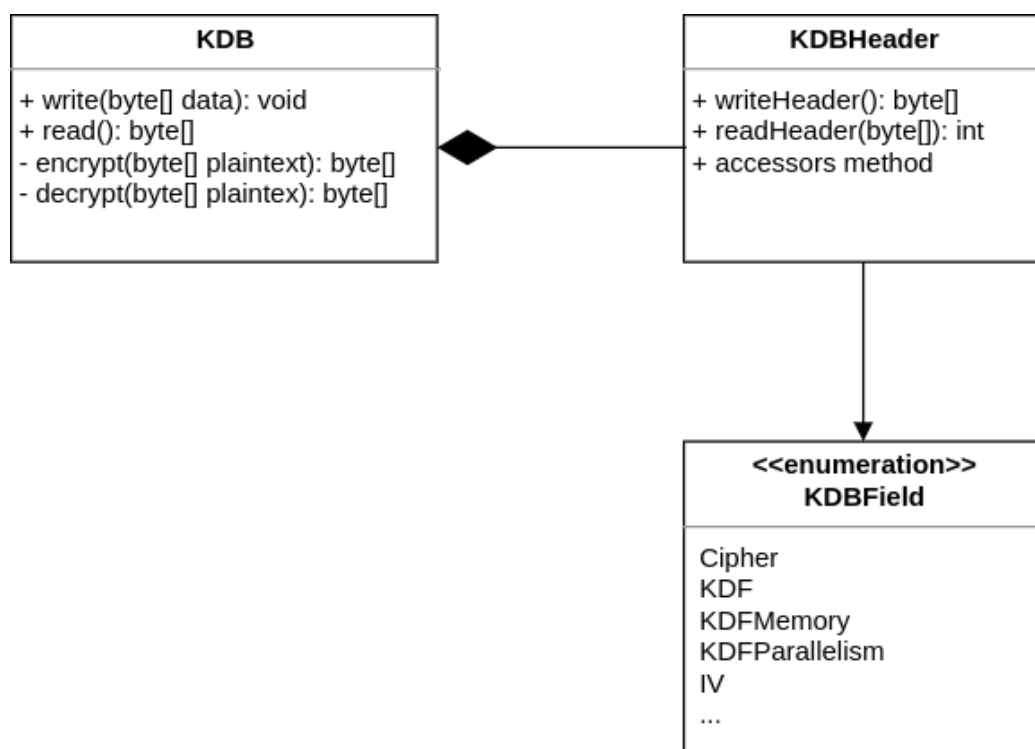


Figura 2.3: rappresentazione UML di KDB

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.

Capitolo 4

Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

Nessuna delle informazioni incluse in questo capitolo verrà utilizzata per formulare la valutazione finale, a meno che non sia assente o manchino delle sezioni obbligatorie. Al fine di evitare pregiudizi involontari, l'intero capitolo verrà letto dai docenti solo dopo aver formulato la valutazione.

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 risulta ottimale. 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. 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.

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 omissis. 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.

Bibliografia

- [1] IETF. Authenticated encryption with aes-cbc and hmac-sha. 2014. <https://tools.ietf.org/html/draft-mcgrew-aead-aes-cbc-hmac-sha2-05>.
- [2] IETF. Chacha20-poly1305. 2018. <https://tools.ietf.org/html/rfc8439>.
- [3] NIST. Aes-gcm. 2007. <https://csrc.nist.gov/publications/detail/sp/800-38d/final>.
- [4] Phillip Rogaway. Authenticated-encryption with associated-data. 2002. <https://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf>.