**Relazione progetto Programmazione a Oggetti**

Carraro Riccardo, mat. 2042346

Toniolo Riccardo, mat. 2042332

**Titolo:** Bernie

**Introduzione**

In un mondo in cui la sicurezza digitale diventa sempre più rilevante e il numero di credenziali di accesso aumenta, possedere un sistema di gestione di credenziali e di memorizzazione sicura di queste, risulta sempre più necessario e utile. Il progetto, infatti, consiste in un applicativo di gestione e salvataggio dati (in particolar modo sensibili) protetti da un’apposita logica di *encrypting* (Vigenère), protetto da un’unica password, che diventa anche la chiave di cifratura utilizzata. In questo modo infatti l’utente, mediante un'unica *Master-Password* è in grado di accedere a tutte le informazioni salvate e con la possibilità di poterle ricercare, visualizzare, modificare o rimuovere, grazie anche ad una interfaccia grafica appositamente strutturata e organizzata, al fine di garantire una quanto più semplice e intuitiva esperienza. L’applicativo permette il salvataggio di elementi quali account, carte di credito, portafogli di cryptovalute,note e contatti.

Particolare attenzione va posta al container utilizzato per la gestione di tali oggetti: il progetto, infatti, implementa una versione rivisitata di un albero rosso-nero binario di ricerca, realizzato principalmente grazie alle conoscenze ottenute durante il corso di “Algoritmi e strutture dati”, che permette di garantire un’alta efficienza in tutte le operazioni di visita, inserimento e eliminazione di elementi a patto di una più complessa implementazioni rispetto ad altri sistemi di container.

L’idea di questo progetto nasce principalmente dalla comodità che un sistema del genere può offrire all’utente, e soprattutto dalla volontà di voler unire conoscenze e concetti interdisciplinari appresi in corsi di “Algoritmi e strutture dati” e “Cybersecurity” sulla gestione e sull’*encryption* dei dati.

**Descrizione del modello**

Il modello logico del progetto è caratterizzato dalle seguenti sezioni fondamentali, quali la gerarchia degli oggetti salvabili, il container utilizzato, il sistema di interfacciamento con il file system e l’interfaccia di gestione.

Il diagramma UML, omesso per motivi di dimensione, è reperibile al seguente [link](https://lucid.app/lucidchart/77a31cfd-fd9f-4709-956e-35a967abd823/edit?page=HWEp-vi-RSFO&invitationId=inv_426cca7e-6ed6-48d1-9905-1d7ff553662e).

Mediante un accurato e quanto più completo sistema di commenti e documentazione all’interno del codice, è possibile dedurre il comportamento dei metodi delle varie classi implementate, le cui definizioni sono di fatto precedute da PRE e POST condizioni, che vanno ad esplicitare il comportamento e il risultato prodotto.

**Gerarchia**

Come previsto dalle richieste implementative, il progetto fa uso di una gerarchia composta da una classe astratta, SerializableObject, e cinque classi derivate concrete, rappresentative dei dati gestiti dall’applicativo. Questi oggetti, derivanti dalla classe base astratta SerializableObject, come suggerisce il nome stesso, implementano un metodo in grado di *serializzare* l’oggetto, fornendo in output una stringa composta da tutte le informazioni dell’oggetto stesso, in modo da permetterne il salvataggio su file (simile a CSV).

La classe SerializableObject, radice della gerarchia, dunque fornisce l’interfaccia base, comprendente i metodi essenziali, che ogni classe derivata dovrà successivamente implementare:

* ***virtual std::string serialize() const***  (virtuale puro) che restituisce una stringa contenente tutte le informazioni del file in formato simil CSV.
* ***virtual bool modify(****const SerializableObject\*****)*** (virtuale puro)che permette la modifica dell’oggetto mediante il passaggio di un puntatore ad un SerializableObject con il quale voglio modificare i dati dell’oggetto corrente.
* ***virtual void accept(****SerializableObjectVisitor\*, bool = false****) const*** (virtuale puro) che permette, sempre su base polimorfa, l’accettazione di un Visitor (secondo il *Visitor Pattern*) per la rappresentazione grafica dell’oggetto.

Decisa infatti la logica di salvataggio dei dati mediante un carattere separatore e un carattere di *escaping*, vengono inoltre definiti e implementati nella classe i metodi atti alla sanitizzazione e alla de-sanittizzazione delle stringhe generate dalla serialize() e della lettura da file, aggiungendo/rimuovendo eventuali escaping e grantendo che la stringa scritta/letta sia nel formato corretto:

* **static std::string sanitize(**const std::string&**)**  che permette di sanitizzare la stringa generata dalla serialize() nel momento in cui i valori inseriti dall’utente nei campi degli oggetti da salvare, contengano esattamente i caratteri utilizzati per *escaping* o come sepratatore.
* ***static std::pair<bool, std::vector<std::string>> deSanitize(****const std::string &****)*** che permette, passata la stringa letta dal file in formato CSV, di restituire una flag di corretta lettura, e un vettore contenente le stringhe necessarie raffiguranti i campi dell’oggetto letto.

La gerarchia dunque, radicata in SerializableObject che possiede il campo *name* come identificativo degli oggetti, prevede le seguenti classi derivate concrete:

* Account, avente i campi relativi a email, username, password.
* CreditCard, avente i campi relativi a proprietario, numero, cvv e data di scadenza.
* Contact, avente i campi relativi a nome, cognome, data di nascita, numero di telefono e email.
* CryptoWallet, avente i campi relativi al nome della blockchain a cui fa riferimento e fino a 24 parole relative ai sistemi di sicurezza utilizzati dai moderni portafogli di cryptovalute.
* Note, avente il campo relativo al testo.

**Classe EncDec\_File per la gestione del file system**

Per l’interazione con il file system per il salvataggio e la gestione dei file per la persistenza dei dati è stata prevista una classe apposita. Essa racchiude dunque tutti i metodi necessari alla scrittura e lettura dei file, occupandosi inoltre, come suggerisce il nome stesso, della procedura di *encryption* e *decryption* delle informazioni. La classe dunque rappresenta l’interfacciamento dell’applicativo con il file system del dispositivo, permettendo una stesura di codice più lineare e pulita nel corpo del programma.

**Interfacciamento con il modello**

Al fine di rendere il codice il quanto più leggibile e mantenibile, la classe del container *RBBSTree* e la classe di accesso alle informazioni sul file *EncDec\_File* sono state logicamente riunite in una classe *Vault* che rappresenta, all’interno del codice, il vero interfacciamento con il *core* dell’applicativo. Inoltre mediante questa scelta è stato possibile ottenere considerevoli vantaggi implementativi, non solo garantendo una più rapida e migliore stesura del codice, ma anche segnando una netta e invalicabile separazione tra il modello logico e grafico, rendendo netta la loro distinzione a livello implementativo.

**Polimorfismo**

Il polimorfismo permea numerose aree del progetto, sia nella parte implementativa della gerarchia, sia nella parte grafica, mediante l’accurata progettazione di classi astratte sia per *Visitors* sia per le stesse pagine.

* **Gerarchia radicata in SerializableObject:**
  + Metodo **modify**: questo metodo permette, come suggerisce il nome, di modificare un oggetto, prendendo come argomento un puntatore a SerializableObject. Ogni classe concreta della gerarchia verificherà mediante un *dynamic\_cast* se il puntatore passato è effettivamente del tipo dinamico corrente, e nel caso in cui lo fosse procede alla modifica, in caso contrario, il metodo, restituisce *false* segnalando che la modifica non è andata a buon fine.
  + Metodo **serialize**: questo metodo si occupa della creazione della stringa contenente tutte le informazioni dell’oggetto da cui viene richiamata. Ogni classe concreta dovrà di conseguenza ridefinire il metodo al fine di produrre la rispettiva stringa per la memorizzazione su file.
  + Metodo **accept**: seguendo il *Visitor Design Pattern* per la realizzazione della parte grafica e dei widget rappresentativi degli oggetti è stato definito un metodo in grado di accettare un visitor, prendendo come parametro un puntatore a SerializableObjectVisitor, classe base astratta dei visitor utilizzati (polimorfismo aggiuntivo).
* **Visitor per la parte grafica:**
  + Data la natura profondamente diversa degli elementi gestiti dalla gerarchia, l’applicativo fa uso di un sistema di *visitor* in grado di poter creare i corretti widget per la rappresentazione degli elementi della gerarchia, predisponendo le rispettive pagine di creazione, visualizzazione e modifica.
  + All’interno della schermata principale del progetto, in cui sono visibili tutti gli elementi salvati, è estato reso possibile, sempre mediante il visitor, assegnare ad ogni elemento la rispettiva icona, migliorando sia dal punto di vista grafico sia dal punto di vista dell’utilizzo l’esperienza finale dell’utente.
  + L’approccio mediante *Visitor Design Pattern* contribuisce a mantenere la logica implementativa e la parte grafica spearate.