|  |  |
| --- | --- |
| **Oggetto** | Relazione progetto Programmazione a Oggetti |
| **Gruppo** | Toniolo Riccardo, mat. 2042332  Carraro Riccardo, mat. 2042346 |
| **Bernie** | Bernie |

**Introduzione**

Bernie è un’applicazione che permette agli utenti di mettere al sicuro i propri dati, utilizzando la tecnica del cifrario di Vigenère. Sostanzialmente l’utente può salvare in questo sistema diverse tipologie di dati, contenenti ad esempio password molto lunghe (molto difficili da ricordare), proteggendole con un’unica master password per l’accesso all’archivio, cosicché al posto di dover ricordare tante informazioni sensibili e rischiose da scrivere semplicemente su un blocco note, ne basta una, la master password, che va a proteggerle.

In un secondo momento queste informazioni salvate saranno reperibili, visualizzabili, modificabili e rimuovibili, ovviamente solo dal possessore della master password.

Le tipologie di oggetti che si potranno salvare sono: account, carte di credito, portafogli di criptovalute, note e contatti.

Il principale punto di forza è la struttura dati container ad alta efficienza e prestazione usata, utilizzante una versione riadattata degli alberi rosso-neri, realizzata da zero, di modo che l’albero possa essere attraversato sia come albero quasi bilanciato (caratteristica peculiare degli alberi rosso-neri), sia come lista doppiamente collegata (che permette ulteriori miglioramenti nella performance dell’albero). In più, tutti gli elementi dell’albero sono sempre mantenuti in ordine, e questo vuol dire che quando poi dovranno essere mostrati all’utente, esso sarà agevolato nella ricerca di quello che gli interessa.

Ho scelto questo progetto perché avevo molta voglia di applicare i concetti imparati nei corsi di Cybersecurity e di Algoritmi e Strutture dati e anche perché i password manager, come Bitwarden o Roboform, sono uno strumento che utilizzo tutti i giorni e che dovrebbe essere utilizzato da tutti per avere una vita digitale più sicura. Spesso però questi software sono complicati e difficili da utilizzare, ed è proprio per questo motivo che abbiamo dedicato molto tempo a rendere chiare, semplici ed intuitive l’interfaccia utente (UI) e l’esperienza utente (UX).

**Descrizione del modello**

Il modello logico si articola in tre parti: la gestione dei vari tipi di oggetto salvabili, la classe per l’uso del filesystem e la classe per il raggruppamento dei queste due sotto un’unica interfaccia per l’utilizzo.

Essendo il diagramma UML per il modello troppo grande, allego qui un link cliccabile per la sua visualizzazione online: [link](https://github.com/RiccardoTonioloDev/Bernie/blob/main/Docs/UML%20class.png?raw=true).

Non andrò nel dettaglio di tutti i singoli metodi del modello in quanto, per tutte le classi appartenenti ad esso, nei vari file *.h* si possono ritrovare assieme alla dichiarazione dei vari metodi nell’interfaccia delle varie classi, dei commenti dettagliati, per ciascuno di essi, che spiegano il funzionamento del metodo desiderato.

**Gerarchia**

In cima alla gerarchia è presente la classe che rappresenta gli oggetti serializzabili (SerializableObject), ovvero convertibili in stringa.

Metodi notevoli riguardanti questa classe (e che poi verranno ereditati dai sottotipi) sono:

* Un metodo **modify** che permette la modifica polimorfa tramite il passaggio di un puntatore di tipo SerializableObject;
* Per arricchire la classe di un ulteriore metodo che rende possibile una soluzione polimorfa per la generazione di interfacce grafiche diverse, basate sul tipo dell’oggetto, questa classe possiede anche un metodo **accept**, che accetta un puntatore a SerializableObjectsVisitor (la classe base astratta per la gerarchia dei visitor);

Sono poi presenti due ulteriori metodi statici:

* Il metodo **sanitize**, il quale data una stringa in input, restituisce una stringa in output sanificata (sostanzialmente va ad aggiungere caratteri di escape (“&”) prima di eventuali caratteri di escape e caratteri separatori (“,”) che potrebbero essere presenti all’interno della stringa rappresentante l’oggetto serializzato, poiché questa dovrà essere salvata su di un file in un formato simile al CSV, e quindi anche successivamente leggibile senza errori interpretativi);
* Il metodo **deSanitize**, il quale andrà a ricevere come argomento una stringa rappresentante un oggetto serializzato in formato CSV (quindi una riga del file), e durante la rimozione dei di escape giusti, andrà a inserire in un vettore di stringhe, le giuste stringhe che comporranno l’oggetto serializzato.

La classe Account eredita da SerializableObject e rappresenta degli account online di cui si deve ricordare email, username e password.

La classe CreditCard eredita da SerializableObject e rappresenta delle carte di credito di cui si devono ricordare il proprietario, il numero di carta, il cvv e la data di scadenza.

La classe Note eredita da SerializableObject e rappresenta delle note testuali che si vuole semplicemente rendere sicure.

La classe Contact eredita da SerializableObject e rappresenta dei contatti di cui si vuole salvare il nome, il cognome, la data di nascita, il telefono e la mail.

La classe CryptoWallet eredita da SerializableObject e rappresenta dei portafogli di criptovalute di cui si vuole salvare, il nome della blockchain di appartenenza e fino a ventiquattro parole (parole che compongono la frase di creazione del portafogli).

**Classe per gestione del file system**

È stata poi implementata una classe per l’interazione con il file system, di modo da avere un’interfaccia utilizzabile, che astraesse la difficolta di dover utilizzare il filesystem, rendendo il codice più snello.

Questa classe è chiamata EncDec\_File, e oltre ad occuparsi di quanto detto, fa queste operazioni criptando in scrittura e decryptando in lettura, appunto tramite il cifrario di Vigenère.

**Classe per l’incapsulamento del modello**

Per rendere poi più facile e intuitivo l’utilizzo dell’albero (RBBSTree), combinato con l’utilizzo di una istanza di EncDec\_File, abbiamo deciso di racchiudere il tutto sotto la classe chiamata Vault.

Questa scelta è stata fatta per due motivi:

* Migliorare l’incapsulamento e l’astrazione, rendendo migliore la developer experience;
* Creare un separamento netto tra la parte del modello e la parte grafica. Racchiudendo quindi tutta la parte di modello e controllo dentro un’unica interfaccia (Vault appunto), non si corrono rischi di poter accidentalmente mischiare la parte grafica con la parte non grafica.

**Polimorfismo**

Il polimorfismo in questo progetto è presente in diverse parti:

* **Nella gerarchia del modello**: per quanto riguarda a SerializableObject:
  + Possiede un metodo **modify** che gestisce la modifica di un oggetto intero, appartenente alla gerarchia di SerializableObject, utilizzando come parametro, un puntatore a SerializableObject. In ogni istanza concreta del metodo, viene fatta la conversione al tipo giusto per quanto riguarda al puntatore, e se la conversione risulta in un *nullptr*, il metodo restituisce *false*, altrimenti attua la modifica e restituisce *true*;
  + Possiede un metodo **serialize** che si occupa della creazione di una stringa di serializzazione rappresentante l’oggetto stesso. Ovviamente ogni tipo di oggetto dovrà implementare il proprio modo di serializzarsi;
  + Possiede un metodo **accept**, che accettando un puntatore del tipo SerializableObjectsVisitor, permette la creazione polimorfa di widgets grafici, basati sul tipo dell’oggetto visitato. Questo pattern è chiamato appunto *Visitor design pattern*.
* **Per quanto riguarda ai visitor**: sono stati utilizzati per:
  + La creazione di widget con icone diverse a seconda del tipo dell’oggetto della gerarchia, nella sezione centrale della home page;
  + La creazione di widget per la creazione, modifica o visualizzazione degli oggetti della gerarchia;
* **Nella gerarchia della *Graphic User Interface* (GUI)**: per quanto riguarda le pagine per la creazione, modifica e visualizzazione degli oggetti della gerarchia:
  + È stata fatta una classe base astratta chiamata PagesInterface, cosicché al posto di dover gestire segnali diversi per la creazione di una nuova pagina diversa per ogni tipo diverso della gerarchia, per quanto riguarda alla pagina che gestisce i segnali, si va localmente a creare una classe derivante da PagesInterface, che verrà poi spedita con un segnale tramite un puntatore a PagesInterface. I segnali che potranno emettere queste sottoclassi saranno comunque noti, poiché presenti anche nell’interfaccia di PagesInterface.

**Persistenza dei dati**

Per la persistenza dei dati viene utilizzato una versione rivisitata del formato CSV. È possibile creare e gestire più archivi diversi.

Le righe sono composte in questo modo:

TYPE,name,campo1,campo2,…

Dove TYPE indica il tipo dell’oggetto dell’attuale riga. Name indica il nome identificativo dell’oggetto, e i successivi campi sono diversi a seconda del tipo.

Voglio specificare che le logiche di serializzazione, de-serializzazione, sanificazione e de-sanificazione sono state scritte interamente da noi, senza fare uso di alcune librerie esterne.

Sono presenti dei file di prova con i nomi:

* prova1\_correct (password: prova1);
* prova2\_fail (password: prova2);
* prova3\_correct (password: prova3);
* prova4\_fail (password: prova4);

i quali possiedono due oggetti per categoria al loro interno, così da poter utilizzare già qualche archivio di esempio.

I file che finiscono con *correct* sono apribili senza problemi con la password giusta, mentre quelli che finiscono con *fail* sono stati corrotti apposta per mostrare che il sistema è in grado di rilevare gravi modifiche manuali al file.

Il programma è stato pensato per non essere utilizzato andando a modificare i file manualmente, ma solo tramite l’interfaccia utente, tuttavia abbiamo ritenuto opportuno andare a cercare di salvaguardare anche questi spiacevoli inconvenienti.

Siccome non è possibile visualizzare il contenuto del file in quanto cifrato (si leggeranno caratteri incomprensibili), abbiamo aggiunto una funzionalità nella *menu bar* chiamata “See decrypted database”, la quale permette (se si è effettuato l’accesso ad un archivio) di visualizzare il contenuto del file in maniera non cryptata.

**Container realizzato**

Abbiamo deciso di utilizzare come tipo di container un albero rosso-nero, che abbiamo chiamato RBBSTree.

L’albero rosso nero permette di avere una complessità ben definita di O(log(n)) in tutti quei metodi che avrebbero complessità definita sull’altezza dell’albero.

Inoltre abbiamo deciso di utilizzare questo tipo di struttura poiché è particolarmente veloce, appunto per quanto appena detto, nella ricerca e nell’inserimento ordinato (appunto entrambi O(log(n))).

Tuttavia per poter utilizzare il container come fosse una lista doppiamente collegata abbiamo fatto si che ogni nodo fosse collegato non solo al genitore, al figlio sinistro e al figlio destro, ma anche al predecessore e al successore, così da poter attraversare l’albero partendo dal minimo (o dal massimo) e andando avanti di successore in successore (o di predecessore in predecessore), tramite appositi iteratori.

Questa modifica tuttavia va a migliorare anche ricerche del minimo nel sottoalbero radicato nel figlio destro di un nodo (poiché sarà il suo successore) e del massimo nel sottoalbero radicato nel figlio sinistro di un nodo (poiché sarà il suo predecessore). Questo rende possibile velocizzare metodi come la rimozione, in quanto questa fa uso appunto di questi tipi di metodi.

Tenere traccia del predecessore e del successore di un nodo in questo modo ha una complessità di tempo costante, e quindi non impatta sulla complessità generale garantita dagli alberi rosso-neri.

Ulteriormente a ciò abbiamo fatto si che l’albero sia una struttura dati generica facente uso di template, e quindi riutilizzabile per qualsiasi tipo si voglia.

**Funzionalità implementate**

Le funzionalità implementate sono, per semplicità, suddivise in tre categorie: funzionali, estetiche e documentative.

Le prime comprendono:

* Gestione di cinque tipologie di dati;
* Conversione e salvataggio in formato CSV criptato tramite cifrario di Vigenère;
* Funzionalità di ricerca tramite nome identificativo;
* Funzionalità di ricerca in base al tipo;
* Funzionalità di gestione di più archivi diversi;
  + Possibilità di crearne nuovi;
  + Possibilità di accedere ad archivi esistenti;
  + Possibilità di effettuare logout dai vari archivi tramite il pulsante “Logout” nella *menu bar*;
* Alta efficienza e prestazioni nella struttura dati ad albero rosso-nero creata (RBBSTree);
* Salvataggio automatico dei dati;
  + Per migliorare l’esperienza utente (UX), abbiamo fatto si che le modifiche agli archivi, di qualsiasi tipo, vengano automaticamente salvate, facendo si che l’utente non debba attivamente preoccuparsi del salvataggio.

Le funzionalità grafiche sono:

* Pulsantiera in basso per filtrare tra gli elementi dell’archivio tramite la ricerca per tipo;
* Campo per la ricerca tramite nome nella parte alta della finestra per filtrare in tempo reale gli elementi dell’archivio tramite nome identificativo;
  + La ricerca avviene utilizzando le sotto sequenze di stringhe, per rendere più intelligente il filtro tramite identificativi;
* Pulsante in alto a sinistra per rimuovere qualsiasi tipo di filtro e ritornare alla visione ordinata degli elementi dell’archivio;
* Richiesta all’utente, in caso di rimozione di un oggetto, di confermare la rimozione tramite una *message box*;
  + Questo per migliorare l’esperienza utente, e renderlo ulteriormente consapevole;
* Richiesta all’utente, in caso di rimozione di un archivio, di confermare la rimozione tramite una *message box*;
  + Sempre per migliorare la UX e rendere l’utente più consapevole;
* Realizzazione dell’intera applicazione in modalità *Single Page Application*, per migliorare la UX e rendere la navigazione all’interno dell’applicazione, più fluida e semplice;
* Possibilità di visualizzare i vari tipi di dato diversi, attraverso diverse interfacce per ogni tipo;
  + È da notare che i campi, se si va a visualizzare i campi dell’oggetto tramite l’apposito bottone (quello con all’interno un occhio), sono selezionabili e copiabili ma non modificabili, senza essere però campi disabilitati, questo per migliorare l’accessibilità alle informazioni (si pensi al caso in cui si vuole copiare una password, precedentemente salvata);
* Utilizzo di icone nei pulsanti;
* Utilizzo di colori e stili grafici;
* Effetti grafici come cambio del colore al passaggio del mouse;
* Non si va ad utilizzare la solita schermata per la selezione, creazione o rimozione di un file, ma si va ad utilizzare una schermata realizzata appositamente per rendere il più semplice possibile il processo di selezione, creazione o rimozione, senza doversi preoccupare di percorsi o directory;

Le funzionalità documentative sono (funzionalità che forniscono documentazione o che sono state fatte per testare parti interne dell’applicazione):

* Presenza nella *menu bar*, di un pulsante “Manual” per accedere al manuale di utilizzo dell’applicazione. Sebbene l’applicazione sia stata studiata per essere più intuitiva possibile, è stato aggiunto il manuale per poter chiarire ulteriormente, in caso di necessità, i dubbi dell’utente;
* Presenza nella *menu* *bar*, di un pulsante “See decrypted database”, il quale permette (se si è effettuato l’accesso ad un archivio) di visualizzare il contenuto del file in maniera non cryptata;
* All’interno della cartella *Testing*, è possibile trovare gli unit test automatizzati, utilizzati per provare in modo estensivo e mirato le parti più sensibili ad errori, quali l’RBBSTree, l’EncDec\_File e il Vault. Per andare ad eseguire questi test basta utilizzare i seguenti comandi in sequenza (I primi due comandi sono di compilazione, mentre il terzo è di esecuzione):
  + Se si sta utilizzando come sistema operativo macOS (faccio questa distinzione con macOS, in quanto questo sistema, se si utilizza qmake, va ad generare un bundle, che non rende possibile l’accesso corretto alle cartelle di salvataggio):
    - cmake CMakeLists.txt
    - make
    - ./Bernie
  + Se si sta utilizzando come sistema operativo una qualsiasi distribuzione di Linux:
    - qmake
    - make
    - ./Bernie

Le funzionalità elencate sono intese in aggiunta a quanto richiesto dalle specifiche del progetto.

**Rendicontazione ore**

|  |  |  |
| --- | --- | --- |
| **Attività** | **Ore Previste** | **Ore Effettive** |
| Studio e progettazione | 10 | 11 |
| Sviluppo del codice del modello | 10 | 17.5 |
| Studio del framework Qt | 10 | 4 |
| Sviluppo del codice della GUI | 10 | 19 |
| Test e debug | 5 | 10 |
| Stesura della relazione | 5 | 5 |
| **totale** | 50 | 66 |

Il monte ore è stato superato per tre motivazioni principali:

* Per quanto riguarda lo sviluppo del container sono successe due cose:
  + Ho dovuto studiare e implementare il metodo di eliminazione dall’albero, in quanto non spiegato a lezione. Questo ha richiesto tempo, soprattutto per via per l’alta complessità dell’algoritmo per “aggiustare” l’albero dopo l’inserimento (circa 2 ore);
  + Ho dovuto re-implementare parti significative della struttura, in quanto prone a determinati tipi di errori in qualche caso specifico, da capo (circa 3 ore);
* Per quanto riguarda all’interfaccia grafica, abbiamo dedicato veramente molto tempo a curare e perfezionare l’esperienza utente e l’interfaccia grafica, nonché ad integrare funzionalità non richieste (circa 6 ore);
* È stato dedicato tempo (circa 4 ore) anche alla creazione di un programma di testing automatizzato (parte del monte ore di Test e debug), per riuscire a poter generare, riprodurre, scovare e sistemare errori con facilità.

Se si togliessero queste ore dal monte ore totale, arriverei ad un nuovo totale di: 51 ore.