Relazione progetto Programmazione a Oggetti

Toniolo Riccardo, mat. 2042332
Carraro Riccardo, mat. 2042346

Titolo: Bernie[1]

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 sarà utilizzata per 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.
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, il cui unico attributo è il campo **name**, che identifica l'oggetto stesso.
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;
- Un metodo **serialize** che permette la costruzione di una stringa, rappresentante l'oggetto di invocazione in formato CSV;
- 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 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 (Le "," appartenenti agli attributi, devono essere considerate come diverse dalle "," usate per separare gli attributi));
- 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 caratteri di escape giusti, andrà a inserire in un vettore di stringhe, le giuste stringhe che comporranno l'oggetto serializzato.

[1] Il nome Bernie è stato tratto dal nome del paguro, facente parte del cartone animato per bambini "Zig & Sharko", noto per la sua astuzia e intelligenza.

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 difficoltà 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 decriptando 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, in quanto la parte grafica andrà a lavorare solamente con Vault, e non interagirà direttamente con nessuno degli elementi interni al modello stesso.

Polimorfismo

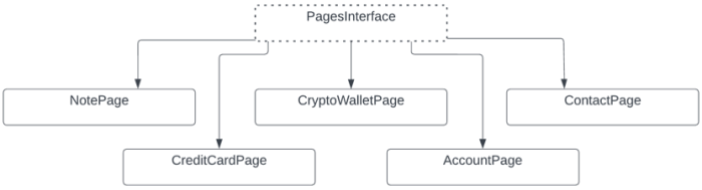
Il polimorfismo in questo progetto è presente in diverse parti:

- **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 tramite *dynamic_cast*, al tipo della classe in cui si sta implementando il metodo concreto, per quanto riguarda al puntatore passato. Se la conversione risulta in un *nullptr*, il metodo restituisce *false*, altrimenti attua la modifica, tramite operatore di assegnazione 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, in base al tipo, degli oggetti della gerarchia;
  - È inoltre da notare che il design pattern del visitor, contribuisce a mantenere una separazione netta tra il modello e l'interfaccia grafica (non andiamo quindi a contaminare il modello con elementi grafici, rimanendo sempre una soluzione polimorfa);
- **Nella gerarchia dei visitor**: per quanto riguarda a SerializableObjectsVisitor:
  - Essendo i vari visitor sottotipi di SerializableObjectsVisitor, si permette di utilizzare un solo metodo **visit** polimorfo da parte di un puntatore del tipo SerializableObjectsVisitor, all'interno di un solo **accept** nella parte della gerarchia del modello, che farà quindi uso del vero **visit** appartenente al tipo dinamico del puntatore utilizzato. In questo modo non dobbiamo fare accept diversi per visitor diversi, grazie appunto all'invocazione polimorfa di **visit**.
- **Nella gerarchia della *Graphic User Interface* (GUI)**: per quanto riguarda le pagine per la creazione, modifica e visualizzazione degli oggetti della gerarchia:
  - Sostanzialmente la pagina che gestisce la selezione del tipo da creare, va a crearsi localmente il widget giusto come form di creazione, a seconda del tipo selezionato. Tuttavia per passarlo tramite un

segnale, utilizza come metodo di passaggio, l'upcast implicito al tipo puntatore a PagesInterface. In questo modo la pagina che dovrà gestire il segnale, non dovrà gestire con cinque slot diversi, cinque possibili segnali (uno per ogni tipo, usato come parametro del segnale), ma gliene basterà uno generico (usante come parametro solo PagesInterface). Inoltre, siccome tutte le classi derivanti da PagesInterface condividono gli stessi segnali (ereditati da PagesInterface appunto), per esprimere la creazione o l'editing di un oggetto, basterà solamente utilizzare i segnali di PagesInterface all'interno degli eventuali connect.

Descrizione della GUI

Per quanto riguarda la GUI (Graphic User Interface), la gerarchia in forma minimale è la seguente:



Queste pagine (per quanto riguarda alla gerarchia di PagesInterface) sono la diretta rappresentazione in formato widget, delle classi appartenenti alla gerarchia del modello.
Il resto delle pagine non dipende direttamente da alcun modello gerarchico, ma sono separate tra di loro, appunto perché rappresentano semplicemente delle pagine per svolgere un singolo compito.
Per quanto riguarda invece la gerarchia dei visitor, questa è stata fatta per due motivi:

- Rende possibile l'aggiunta di un solo metodo **accept** alle classi appartenenti alla gerarchia di SerializableObject, facendo uso di un metodo virtuale **visit** da parte del visitor (che andrà a richiamare il vero metodo virtuale del tipo dinamico), senza la necessità di dover fare due accept diversi;
- I due visitor comunque posseggono una struttura comune, e quindi abbiamo deciso di utilizzare una classe da cui farli derivare, per avere le firme dei metodi già scritte, e di conseguenza si è solo implementato il corpo.

Abbiamo basato l'uso dell'applicazione e il flusso di utilizzo su uno storyboard precedentemente creato. Tuttavia la sua dimensione è estremamente grande, quindi lascio qui un link per poterlo visualizzare: link.

Persistenza dei dati

Per la persistenza dei dati viene utilizzato una versione rivisitata del formato CSV. È da notare la possibilità di creare e gestire più archivi diversi, e non uno solo.
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 nessuna libreria esterna.
Sono presenti dei file di prova con i seguenti 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" (Dentro l'action "File"), la quale permette (se si è effettuato l'accesso ad un archivio) di visualizzare il contenuto del file in maniera non criptata.

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, cancellazione e nell'inserimento ordinato (appunto O(log(n)) a livello di complessità).
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. Tenere traccia del predecessore e del successore di un nodo, ha una complessità di tempo costante, e quindi non impatta sulla complessità generale garantita dagli alberi rosso-neri.

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), portando questo tipo di ricerca a complessità costante. Questo rende possibile velocizzare metodi come la rimozione di un nodo dall'albero, in quanto questa fa uso appunto di questi metodi.

Inoltre, per aumentare ulteriormente l'efficienza abbiamo fatto uso di metodi iterativi e non ricorsivi per lo scorrimento dell'albero.
L'accesso al minimo e al massimo dell'albero intero avviene in tempo costante, in quanto sono elementi salvati all'interno della struttura.

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 filtraggio 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" (Dentro l'action "File") nella *menu bar*;
    - Possibilità di rimuovere archivi esistenti;
- 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, cosicché l'utente non debba attivamente preoccuparsi del salvataggio.

Le funzionalità grafiche sono:

- Pulsantiera nella parte bassa della finestra, per filtrare per tipo gli elementi dell'archivio;
- Campo per la ricerca tramite nome nella parte alta della finestra per filtrare in tempo reale (senza dover premere bottoni per avviare la ricerca) gli elementi dell'archivio tramite nome identificativo;
    - La ricerca avviene utilizzando le sotto sequenze di stringhe, per rendere più intelligente la ricerca per nome identificativo;
- Pulsante in alto a sinistra per rimuovere qualsiasi tipo di filtro e ritornare alla visione ordinata degli elementi dell'archivio (Il pulsante "All", con affianco l'immagine di una casa);
- 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 volesse copiare una password, precedentemente salvata);
- Utilizzo di icone nei pulsanti;
- Creazione di un logo su misura da essere utilizzato come logo dell'applicazione;
- Utilizzo di icona con angoli arrotondati per l'applicazione;
- Utilizzo di colori e stili grafici;
- Effetti grafici come cambio del colore al passaggio del mouse;
- Non si va ad utilizzare la solita finestra di dialogo 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" (Dentro l'action "File") 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" (Dentro l'action "File"), 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, scritti da me (senza l'utilizzo di librerie esterne), 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) (Bisogna prima ovviamente posizionarsi all'interno della cartella *Testing* con il terminale):
    - Se si sta utilizzando come sistema operativo MacOS (faccio questa distinzione con MacOS, in quanto questo sistema, se si utilizza QMake, andrà a 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 (ad esempio la macchina virtuale fornita per testare il codice):
        - qmake
        - make
        - ./Bernie
- Ogni classe appartenente al modello possiede per ogni metodo, della documentazione dettagliata sotto forma di commenti multi-riga, per renderne più chiaro l'utilizzo tramite una descrizione e post e precondizioni del metodo stesso.

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

Rendicontazione ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	8	11
Sviluppo del codice del modello	14.5	15
Studio del framework Qt	5	4
Sviluppo del codice della GUI	12.5	18
Test e debug	5	10
Stesura della relazione	5	5
totale	50	63

Il monte ore è stato superato per tre motivazioni principali:

- Per quanto riguarda lo sviluppo del container ho dovuto studiare e implementare il metodo di eliminazione dall'albero, in quanto non spiegato nel corso di algoritmi. Questo ha richiesto tempo, soprattutto per via dell'alta complessità dell'algoritmo per "aggiustare" l'albero dopo l'eliminazione (circa due ore);
- Per quanto riguarda all'interfaccia grafica, abbiamo dedicato veramente molto tempo a curare e perfezionare l'esperienza utente e l'interfaccia utente, nonché ad integrare funzionalità aggiuntive (circa sei ore);
- È stato dedicato tempo (circa quattro ore) anche alla creazione di un programma di testing automatizzato (non richiesto), per riuscire a poter generare, riprodurre, scovare e sistemare errori con facilità.

Divisione dei compiti

Per quanto riguarda la parte di codice scritta da me, io sono responsabile per le seguenti parti:

- Classi della gerarchia:
  - SerializableObject;
    - Tutto tranne la funzione deSanitize;
  - Tutto CryptoWallet;
  - Tutto Account;
- Classi aggiuntive:
  - Tutta la classe Telephone;
- Albero rosso-nero:
  - Funzione transplant;
  - Funzione deleteT;
  - Funzione deleteFixUp;
  - Funzione searchNode;
  - Il distruttore profondo;
  - Tutto l'iteratore;
  - Funzioni begin e end;
- EncDec_File:
  - Tutto tranne la funzione decFromFile;
- Vault:
  - Funzione loadFromStorage;
  - Setup della classe e del costruttore;
  - Funzione loadStorage;
  - Funzione searchSerializableObjects;
  - Funzione filteredVectorize;
  - Funzione modifyTreeObj;
  - Funzione isInitialized;
- Tutto il main per il testing, e quasi tutte le funzioni per il testing;
- Parte grafica:
  - Sviluppato assieme al compagno (abbiamo lavorato, solo in questa parte, sullo stesso codice contemporaneamente, mediante la funzionalità "Code with me" offerta dall'IDE CLion (utilizzato per la scrittura del codice)), le seguenti pagine e quindi relative classi:
    - CreateDBPage;
    - DBSelectedPage;
    - DBSelectedToRemovePage;
    - HomePage;
    - LandingPage;
    - MainWindow;
    - SelectDBPage;
    - SelectDBToRemove;
    - TypeSelectionPage;
    - SerializableObjectsVisitor;
    - PagesInterface;
  - Sviluppato da solo le pagine e classi:
    - AccountPage;
    - CryptoWalletPage;
    - VisitorManageItem;
    - VisitorListItem;
  - Sviluppato da solo i seguenti componenti:
    - MenuButton;
    - TypeSelectionButton;

Compilazione ed esecuzione

È stata resa possibile la compilazione del progetto tramite due metodi (prima di eseguire i comandi, collocarsi nella cartella src, tramite il terminale):

- QMake (Da usare su distribuzioni Linux): usando i seguenti comandi in successione:
  - qmake
  - make
- CMake (Da usare su MacOS): usando i seguenti comandi in successione:
  - cmake CMakeLists.txt
  - make

L'esecuzione di questi comandi porta alla compilazione del progetto con destinazione *./Bernie*, dove "." è la cartella dove è presente il file *.pro* .
Per eseguire l'eseguibile quindi, una volta aver completato i precedenti due comandi, basterà usare il comando:

./Bernie

È stato fornito anche il metodo di compilazione tramite CMake nel caso si volesse compilare il progetto su una macchina con sistema operativo MacOS, in quanto QMake, solo nel caso di MacOS, va a generare una serie di sottocartelle con l'eseguibile all'interno, e non nella cartella corrente (e questo determinerebbe un errore nell'accesso alla lettura dei files nella cartella dedicata al loro salvataggio, da parte dell'eseguibile).

Note finali

Lo sviluppo del progetto è stata una bella attività dal punto di vista della crescita del mio bagaglio culturale. Abbiamo deciso apposta io e il mio compagno di fare un progetto di gruppo e non singolo, semplicemente per poter applicare tecniche organizzative ed amministrative per lo sviluppo del progetto.
Non si sono verificate difficoltà durante il percorso, se non solamente nella parte relativa al capire come funziona Qt a livello di segnali e di slot, poiché mai utilizzato in precedenza (che comunque si è risolta velocemente, grazie alla notevole quantità di informazione online nei forum ufficiali, documentazioni, e StackOverflow).