PROGRAMMAZIONE AD OGGETTI
RELAZIONE PROGETTO
“Liceo scientifico Richard Feyman”

Cognome: Bile
Nome: Ezanin Christian Prince Carlos
Matricola: 1096677

INDEX

I) Prima di iniziare

1) Ambiente di sviluppo

Sistema operativo: Ubuntu 16.04 32-bit
- Compilatore: GCC 5.2.1
- Versione libreria Qt : 4.8

Sistema operativo: Windows 10 Pro 32-bit
- Compilatore: MinGW 5.3 .0 (32-bit)
- Versione libreria Qt: 5.6

2) Ore impiegate per il progetto: 152 ore

Modello: 40 ore (7 ore di progettazione + 33 ore di codifica)
Controller: 4 ore (1 ora di progettazione + 3 ore di codifica)
View: 95 ore (11 ore di progettazione + 84 ore di codifica)
Debug: 13 ore (incluse le ore per il memory leak analyser)

Tutte queste ore di lavoro superate sono dovute alla grafica e alla mancanza di idee di un progetto più semplice.

3) Nota aggiuntiva

Il programma contiene tante classi, tante di quelle sono classi della GUI, create appositamente per semplificare il lavoro. In questa relazione spiegherò le classi più importanti.

II) Scopo del progetto

Questo progetto ha lo scopo di creare un programma che permetta ad un insieme ristretto di utenti di un liceo scientifico, di interagire tra loro. Gli utenti considerati sono : Il presidente, il segretario, il professore, lo studente, l'amministratore sistema e l'amministratore biblioteca. Il programma deve offrire:

- ➔ una biblioteca a tutti gli utenti, dove potranno cercare e leggere un libro che desiderano.
- ➔ Un servizio email tra impiegati (Presidente, segretario e professore) .
- ➔ La possibilità per i professori di assegnare compiti/esercizi in formato xml e/o pdf ai loro studenti
- ➔ Forum gruppi per l'interazione tra studenti tramite i post che pubblicheranno, e la possibilità per loro di visualizzare i compiti assegnati dai professori, di farli al momento se sono compiti in formato xml.
- ➔ Gestione della biblioteca, utenti e del sistema

I privilegi degli utenti, oltre a l'usufruito della biblioteca, sono riassunti nella seguente tabella:

	Privilegi
Amministratore sistema	-Aggiunge/ rimuove presidente. -Crea/elimina/ Modifica gruppi. -Visualizza/elimina i post nei gruppi.
Amministratore biblioteca	-Gestione totale dei libri della Biblioteca. -Gestione totale delle categorie libri. -Gestione dei libri in primo piano.
Presidente	-Aggiunge/rimuove professore e segretario. -Assegna/toglie classe a professori. -Aggiunge/rimuove classe. -Aggiunge/ rimuove sessione ed indirizzo classi. -Può inviare/ricevere mail a/da impiegati, e può cancellare le proprie mail.
Segretario	-Aggiunge/rimuove/modifica info studenti. -Può inviare/ricevere mail a/da impiegati, e può cancellare le proprie mail.
Professore	-Assegna/rimuove compito alle classi che insegna. -Può inviare/ricevere mail a/da impiegati, e può cancellare le proprie mail.
Studente	-Iscriversi/annullare iscrizione, a gruppi. -Scrivere nuovi / visualizzare post su gruppi in cui è iscritto. -Gestione dei suoi post pubblicati nel gruppo. -Visualizzare/fare i compiti assegnati dai professori, e visualizzare la correzione del professore. -Può vedere foto profilo di studenti iscritti nello stesso gruppo suo.

Oltre a questi privilegi, tutti gli utenti hanno la possibilità di cambiare la propria password, e gli utenti non amministratori possono cambiare la loro foto profilo.
Per aiutare l'utente ad usare il programma, ho implementato delle funzionalità di ricerca che variano a seconda dei privilegi dati, esempio: ricerca studente per i segretari, ricerca libro per tutti gli utenti ecc…

III) File e Database

1**) Formati File**

Gli utenti hanno bisogno di leggere e/o modificare file, il mio programma permette tali operazioni su due formati di file: .xml (per lettura e scrittura), .pdf (per lettura).

- **File xml:**

Ho creato la classe filexml per una migliore gestione dinamica dei file . Un oggetto della classe filexml rappresenta un generico file .xml ben formato, senza nodi attributo. L'oggetto ha diversi metodi pubblici che possono essere chiamati per la ricerca, lettura e modifica dei nodi con possibilità di salvare le modifiche apportate. Per poter dare risultati precisi e corretti, la classe ha la seguente precondizione : Tutti i nodi elemento devono avere un nome diverso dal padre. Su filexml ho fatto la scelta di usare la copia senza condivisione di memoria, piuttosto che usare gli smartpointer in quanto complicherebbe molte operazioni di modifica su file. Una scelta onerosa in termini di memoria, sapendo che un nodo pesa circa 100 byte nella classe, ma vantaggiosa per le operazioni che voglio fare.

- **File pdf**

Per i file pdf uso la classe QDesktopServices fornita dalla libreria Qt. Tale classe chiama un programma, preinstallato sulla macchina dell'utente, in grado di leggere file .pdf, se non esiste tale programma dà un messaggio di errore.

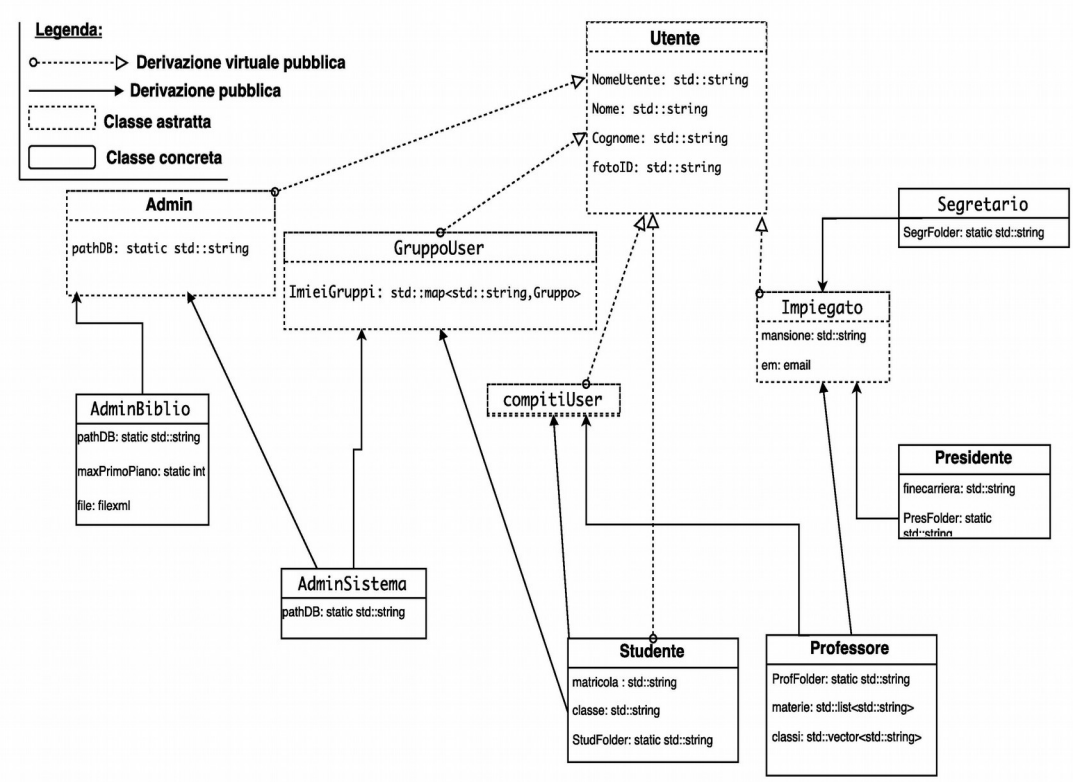
2)Database

Tutto il database è gestito con file xml grazie alla classe filexml. Per evitare di creare una nuova classe che mi rappresenta il database, ho inserito una variabile statica privata di tipo string nella classe filexml, che contiene il percorso verso la cartella del database.

IV) Contenitore

Tra le classi del modello, c'è la classe template Container, che ho creato appositamente per contenere una lista di file in formati diversi. Dispone dell' iteratore costante e non costante per percorrere la lista ed effettuare varie operazioni. La differenza tra Container e gli altri contenitori della libreria standard, è che Container dispone del metodo pubblico overloadato void AddFileDaLista(const filexml& f, const string& tagname) che permette di caricare i file di tipo T, elencati con i tag tagname nel file xml f. Uso principalmente Container per questa sua capacità, mentre uso gli altri contenitori della libreria standard per altre operazioni.

V) Gerarchia delle classi del modello



- **Utente:**

Rappresenta un generico utente della scuola. E' caratterizzato da un nome, un cognome, un nome utente e una fotoprofilo.

Metodi virtuali puri	Metodi virtuali non puri
bool *esiste*() const: restituisce true se l'utente connesso è registrato, altrimenti restituisce false. **Utente* *clone*() const:** restituisce un puntatore alla copia dell'utente connesso. **bool *VerificaPassword*(const string& p) const :** Verifica se p è password dell'utente connesso **list<string> *DaiDatiPersonali*() const:** restituisce una lista di informazioni sull'utente connesso. **bool *CambiaFotoProfilo*(const string& path)const:** restituisce true se il cambio foto profilo ha auto successo. **bool *CambiaPassword*(const string& newPass)const:** restituisce true se il cambio password ha avuto sucesso.	**list<string> *daiMaterieInsegnate*()const:** restituisce una lista delle materie insegnate nella scuola. **vector<string> *daiClassi*()const:** restituisce i nomi di tutte le classi presenti nella scuola

- **GruppoUser:**

Rappresenta un utente di gruppi forum. E' caratterizzato da un insieme di gruppi ai quali è iscritto.

Metodi virtuali non puri
void *InitialiseGruppi*(): Inizializza la lista dei gruppi ai quali è iscritto. **string *getGroupMemberFotoProfile*(const string& nu = "")const:** restituisce il percorso della foto profilo di nu.

- **compitiUser:**

Rappresenta un utente che visualizza e/o crea e/o modifica compiti per studenti.

Metodi virtuali non puri
vector<string> *codiciCompiti*()const: restituisce i codici (chiave univoca) di tutti i compiti/esercizi creati da tutti i professori del liceo. **vector<string>*giveCompitiTitle*()const:** restituisce i titoli di tutti i compiti/esercizi creati da tutti i professori del liceo. **vector<string> *giveCompitiDates*()const:** restituisce le date di assegnazioni, in ordine crescente di creazione, di tutti i compiti/esercizi creati da tutti i professori del liceo. **bool *commentaRispCompito*(const string& codice, int risposta, const string &commento) const:** restituisce true se il commento "commento" all'elaborato "risposta" del compito "codice", è stato aggiunto, altrimenti restituisce false.

- **Admin**

Rappresenta un generico amministratore. E' caratterizzato da un percorso verso la cartella privata degli amministratori.

Overridings
bool *esiste*() const; **Utente **clone*() const;** **bool *VerificaPassword*(const string& p) const;** **list<string> *DaiDatiPersonali*() const;** **bool *CambiaFotoProfilo*(const string &path) const;** **bool *CambiaPassword*(const string &newPass) const;**

- **AdminBiblio**

Rappresenta amministratore della biblioteca. E' caratterizzato da un percorso verso la sua cartella privata e da un numero limitato di libri che può mettere in primo piano.

Overridings
bool *VerificaPassword*(const string& p) const; **bool *esiste*() const;**

- **AdminSistema:**

Rappresenta un amministratore sistema. E' caratterizzato un percorso verso la sua cartella privata.

Overridings
bool *VerificaPassword*(const string& p) const;

```
bool esiste() const;
```

- **Impiegato:**

Rappresenta un impiegato della scuola. E' caratterizzato dalla mansione che occupa e dalle sue mail

Overridings

```
list<string> DaiDatiPersonali()const;
```

- **Presidente**

Rappresenta un preside della scuola. E' caratterizzato dalla data di fine carriera e dal percorso verso la sua cartella privata

Overridings

```
bool esiste() const;
Presidente* clone() const;
bool VerificaPassword(const string& p) const;
bool CambiaFotoProfilo(const string &path) const;
bool CambiaPassword(const string &newPass) const;
list<string> DaiDatiPersonali()const;
```

- **Professore**

Rappresenta un professore della scuola. E' caratterizzato un percorso verso la sua cartella privata, dalle materie e dalle classi che insegna.

Overridings

```
bool esiste() const;
Professore* clone() const;
bool VerificaPassword(const string& p) const;
bool CambiaFotoProfilo(const string &path) const;
bool CambiaPassword(const string &newPass) const;
list<string> daiMaterieInsegnate()const;
vector<string> daiClassi()const;
vector<string> codiciCompiti()const;
vector<string> giveCompitiTitle()const;
vector<string> giveCompitiDates()const;
bool commentaRispCompito(const string &codice, int risposta, const string &commento) const;
```

- **Segretario**

Rappresenta un segretario della scuola. E' caratterizzato un percorso verso la sua cartella privata.

Overridings

```
bool esiste() const;
bool VerificaPassword(const string& p) const;
bool CambiaFotoProfilo(const string &path) const;
bool CambiaPassword(const string &newPass) const;
Segretario* clone() const;
```

- **Studente**

Rappresenta uno studente della biblioteca. E' caratterizzato un percorso verso la sua cartella privata, dal numero di matricola e dalla classe che frequenta.

Overridings

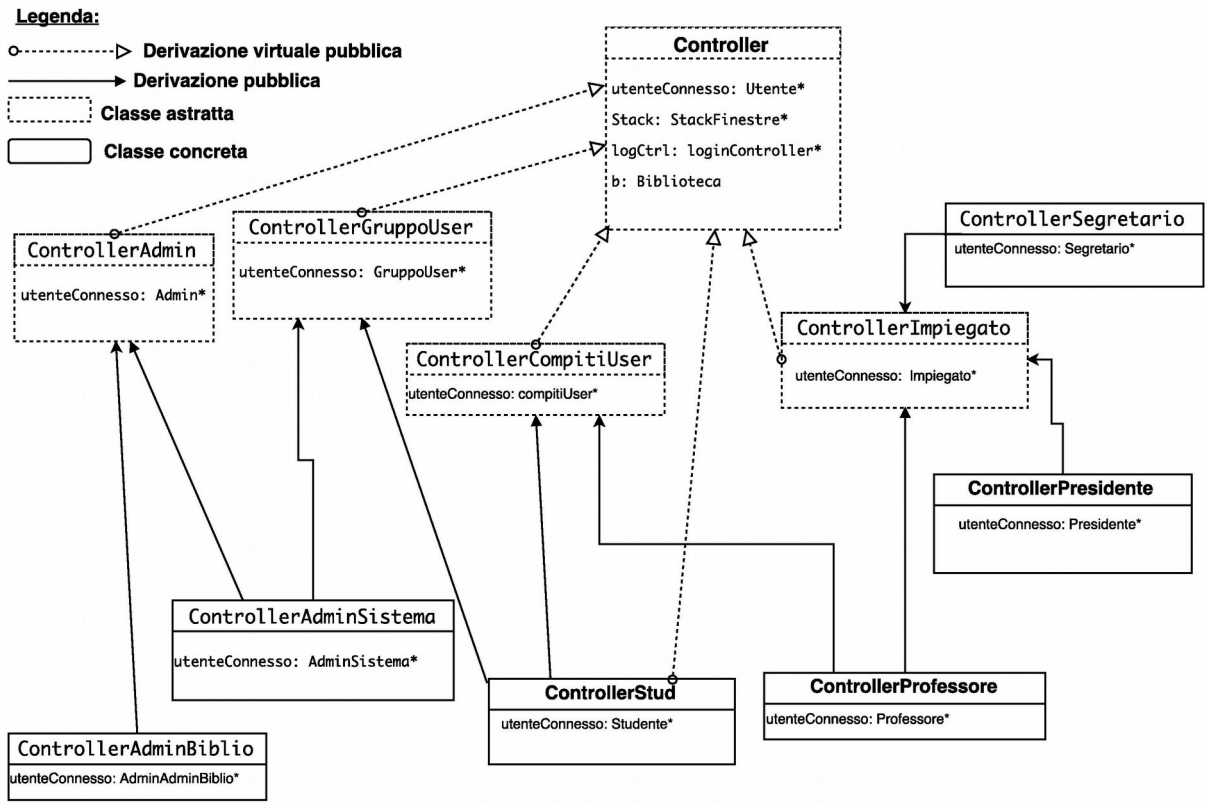
```
bool esiste() const;
bool VerificaPassword(const string& p) const;
bool CambiaFotoProfilo(const string &path) const;
bool CambiaPassword(const string &newPass) const;
Studente* clone() const;
list<string> daiMaterieInsegnate()const;
vector<string> giveCompitiTitle()const;
vector<string> giveCompitiDates()const;
vector<string> codiciCompiti()const;
```

string getGroupMemberFotoProfile(const string &nu)

- **Descrizione del codice polimorfo del modello**

Ogni utente ha dei privilegi che restringono l'insieme di file e cartelle che gli sono permessi. I metodi virtuali puri e non puri, overridati nelle classi derivate, assicurano che le operazioni verranno fatte in base ai privilegi e informazioni dell'utente connesso. Esempi: Nella classe Utente, il metodo *CambiaFotoProfilo* ha accesso solo ai file privati dell'utente connesso, se viene invocato cambierà la foto profilo dell'utente connesso e non di un altro utente. Sempre nella classe utente, se il metodo *daiMaterieInsegnate*() viene invocato da un utente Professore, c'è una forte probabilità che voglia sapere le materie che insegna, e non tutte le materie insegnate nella scuola come prevede l'implementazione di default, per questo viene overridato nella classe professore. Nella classe GruppoUser il metodo virtuale *getGroupMemberFotoProfile* garantisce che solo gli studenti potranno vedere le foto profilo dei compagni nei gruppi forum, mentre l'amministratore di sistema non ha questo privilegio in quanto eredita l'implementazione di default che non fornisce la foto profilo dell'utente. Nella classe compitiUser, tutti i metodi virtuali permettono agli studenti di visualizzare/modificare i compiti assegnati dai professori della classe che frequentano, mentre permettono ai professori di visualizzare/modificare i compiti che assegnano alle classi di insegnamento. Insomma, questi metodi virtuali permettono ai controller di eseguire delle chiamate polimorfe in base all'utente connesso per soddisfare determinate richieste.

VI) Gerarchia delle classi Controller



Tutti i controller sono stati creati per tradurre i dati che viaggiano tra modello e view, e anche per aggiornare le finestre. Ogni controller è caratterizzato da un puntatore all' utente connesso con il quale comunica.
La classe base virtuale e astratta, **Controller,** di tutti i controller, ha, oltre al puntatore all'oggetto utente:

- ➔ Un puntatore alla pila di finestre (StackFinestre) aperte dall'utente L'utente potrà aggiungere (aprire) una nuova finestra chiamando il metodo pubblico void ShowNewWindow(QWidget* w) const, un metodo ereditato da tutte le classi derivate. Potrà inoltre tornare alla finestra precedente chiamando il metodo non costante void getPrevFinestra(), questo metodo porta alla finestra precedente cancellando quelle successive nella pila.
- ➔ Un puntatore a loginController, logCtrl. La classe logController non appare nella gerarchia in quanto non è intermediario di nessun oggetto utente e view, ha solo il compito di logare l'utente e di assicurarsi che tutto sia cancellato quando viene chiusa la finestra. L'oggetto StackFinestre puntato da Stack nei campi dati di

Controller, è di proprietà della loginController puntata da logCtrl. Questo perché mi permette di gestire meglio la logout dell'utente e la chiusura della finestra. Infatti la finestra di login, la prima ad essere nella pila, è conosciuta solo dalla loginController. Il metodo della logout void signout() di Controller, chiama, grazie al puntatore logCtrl, il metodo signout() della logController che mostra la finestra di login svuotando lo stack e cancellando il controller dell'utente, che cancella a sua volta utenteConnesso e b (biblioteca).

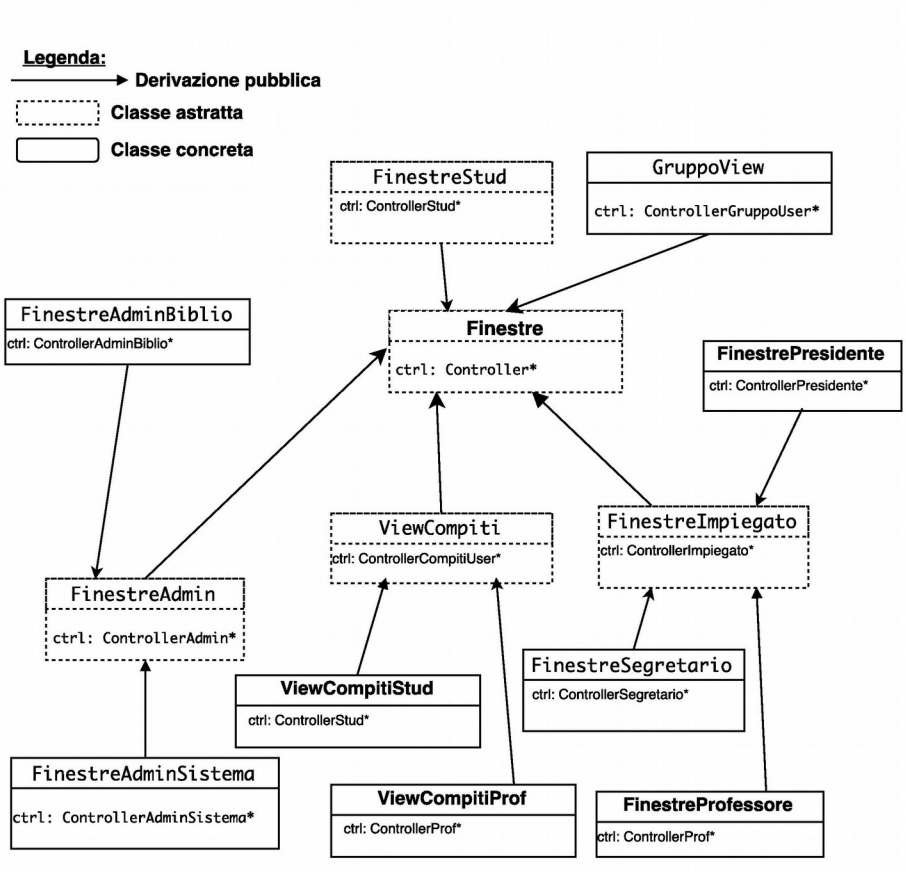
➔ Un oggetto di tipo biblioteca. Visto che tutti gli utenti hanno accesso alla biblioteca, l'ho messo nei campi dati di Controller per evitare implementazioni e chiamate ridondanti a metodi.

➔ Il distruttore virtuale

- **Descrizione del codice polimorfo dei controller**

Controller ha i seguenti metodi virtuali puri implementati dalle classi concrete:
bool *inizializzaUtente*(const QString& u) : restituisce true se l'utente connesso u esiste ed è stato inizializzato, altrimenti false.
bool *CheckPassword*(const QString& p) const: Controlla se la password dell'utente connesso è p.
void *HomePage*() : Mostra la homepage dell'utente connesso.
list<string> *CampiDatiProfile*() const: restituisce i campi info del profilo dell'utente.
Questi metodi permettono alle view di effettuare delle chiamate polimorfe su un puntatore a Controller.
La classe **ControllerGruppoUser**, ha un metodo virtuale: void *HomePageGroup*(const QString& nomeGruppo), che mostra la homepage del gruppo nomeGruppo a seconda che si tratti di un gruppo per studenti o di un altro tipo di gruppo che può essere implementato in futuro.
I controller effettuano anche delle chiamate polimorfe sul modello. I costruttori delle classi derivate hanno delle chiamate esplicite alla classe base, alla quale passano, oltre a loginController e lo stack finestre, il puntatore all'utente connesso per eseguire delle chiamate polimorfe sull'utente connesso. Esempio: se l'utente connesso è il Professore, il puntatore utenteConnesso di ControllerProf avrà tipo statico = tipo dinamico = ControllerProf*, mentre i puntatori a utente connesso delle classi ControllerCompitiUser, ControllerImpiegato e Controller avranno tipo dinamico = ControlerProf* diverso dal tipo statico.

VII) Gerarchia principale delle classi View



Qt non permette l'ereditarietà a diamante, perciò la gerarchia delle finestre assomiglia ad un albero dove ogni nodo può avere più di un figlio. La gerarchia delle finestre è grande, qua sopra è riportato l'albero delle finestre principali. Notare che dalle classi nodi foglia derivano altre classi, esempio dalla classe FinestreStud derivano tutte le finestre dell'utente studente. Le finestre principali sono caratterizzate da un puntatore al rispettivo controller con il quale comunicano. Sono state create per essere la base di un insieme di sotto finestre di un certo utente, al fine di permettere una maggior estensibilità del codice.
Finestre è la classe base astratta di tutte le finestre, tranne delle classe login (view che deriva direttamente da Qwidget, implementata per loginController), è astratta perché ha i seguenti metodi virtuali puri:
void *Header*(): Metodo protetto. Costruisce la parte header della finestra.
void *BodyAndFooter*():Metodo protetto. Costruisce le parti Body and Footer della finestra.
void *reloadWindow*(): Metodo pubblico. Ricarica la finestra corrente.
La classe deriva pubblicamente da QWidget.

- **Descrizione del codice polimorfo della view**

Oltre ad ereditare tutto il polimorfismo della libreria Qt, alcune delle finestre che ho progettato sono dotate di metodi virtuali puri e non puri che permettono all'utente della view di eseguire delle chiamate polimorfe su un puntatore alla classe base con tipo dinamico uguale ad una delle classi derivate concrete. Esempio: La funzione reloadWindow() implementata nelle classi concrete, derivanti direttamente o indirettamente da Finestre, permette all'utente di ricaricare la finestra corrente, un uso del metodo si vede nel metodo void ReloadAllWindows() della classe Controller. In più le finestre eseguono delle chiamate polimorfe sui metodi dei controller, con un meccanismo simile a quello dei controller sui modelli.

VIII) Manuale utente
Il programma non richiede un manuale utente.

IX) Nomi utenti e password per prove
- **Presidente**

nome utente: preside
password: presid
- **Professore**

nome utente: prof1
password: profe1
- **Segretario**

nome utente: segr1
password: segre1
- **Studente**

nome utente: ebile
password: pao2017
- **Admin sistema**

nome utente: admin
password: admin
- **Admin biblioteca**

nome utente: adminbiblio
password: adminbib

X) Materiale consegnato
- file .pro di nome progetto-pao-2017.pro . La compilazione del programma dipende fortemente da questo file, non deve perciò subire modifiche.
- File .h e .cpp delle classi.
- Il database fatto di file .xml nella cartella Database. Nella stessa cartella sono presenti dei file pdf che rappresentano libri o compiti per il test del programma.
- File .qrc che contiene il percorso a immagini e loghi necessari alla compilazione del programma.
- La relazione di 8 pagine in formato pdf.