MEDIARY

PROGRAMMAZIONE AD OGGETTI

Relazione del progetto

Anno accademico 2015/2016

Zecchin Giacomo (1070122)

9 settembre 2016

Indice

1 Scopo del progetto

Mediary è il risultato di questo progetto per il corso di programmazione ad oggetti che aveva come scopo quello di creare un applicativo sviluppato in C++/Qt.

L'applicazione intende rappresentare uno spazio personale nel quale annotare tutte le serieTv o i film già visti, preferiti oppure col desiderio di vedere in futuro.
Mediary consente quindi di accedere singolarmente come utente per aggiungere nuovi *media* al proprio diario (*diary*).
L'unione di queste due parole, **Media** e **Diary**, va difatti a creare il nome dell'applicazione.

Il progetto è semplice ma allo stesso tempo funzionale, diretto e comprensibile per l'utente; vediamo nel seguito tutte le funzionalità.

2 Funzionalità offerte

2.1 Utenti

Avviata, l'applicazione presenta subito una finestra centrale per l'inserimento diretto dei dati necessari ad effettuare il **login** ed accedere immediatamente all'area personale.
In alternativa, se non si possiedono già delle credenziali, è presente un bottone per la **registrazione** poco più in basso, che farà entrare nella finestra dedicata alla creazione di un nuovo utente. In ogni caso sono presenti dei controlli di consistenza dei dati in tutte le viste, che sono descritte più approfonditamente nella sezione dedicata **[View→]** .
E' comunque data la possibilità di **modificare i propri dati** in un secondo momento.

2.2 Media

Le funzionalità offerte per i media sono invece quelle di **creazione, modifica, visualizzazione per tipo ed eliminazione**. (In dettaglio nella sezione **[Model→]**)

3 Schema Logico

Il *Design Pattern* utilizzato per la progettazione è il modello **MVC**, che separa il *Model* dalla *View* con l'ausilio di due classi che formano il *Controller*.

Vediamo nelle sottosezioni seguenti le tre componenti e le classi che li compongono.

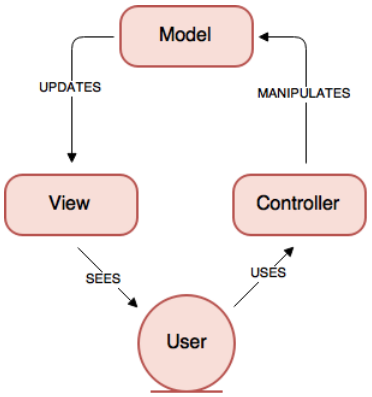

Figura 1: Modularità del pattern MVC

3.1 Model

Di seguito tutte le entità che vanno a formare la parte del *MODEL* del progetto.

3.1.1 Gerarchia polimorfa

La **gerarchia G polimorfa** che contribuisce a formare il modello dati del progetto è composta dalle classi:

- **Media**: E' la classe base astratta della gerarchia polimorfa contenente i campi dati `title, year, creationDate` e `changeDate` comuni a tutte la classi derivate. All'interno è dichiarato il metodo *virtuale puro*

```
virtual void saveMedia(QXmlStreamWriter& xmlWriter) const=0;
```

  per eseguire il **salvataggio POLIMORFO dei media** nel mediaDatabase.xml . ed il metodo *virtuale puro*

```
virtual QString getType() const=0;
```

  che **ritorna il tipo POLIMORFO del media** quando si carica la tabella per visualizzarli tutti nella view apposita (descrizione più avanti nella View).

- **SerieTV**: E' una classe concreta derivata pubblicamente dalla base perché implementa il metodo virtuale puro della base.
  Contiene i campi dati privati aggiuntivi caratteristici di una serieTv `descriptionEp, season, numberEp, lenghtEp` rappresentanti in ordine descrizione, numero della stagione dell'ep., numero episodio e lunghezza in minuti.

- **Film**: E' una classe concreta derivata pubblicamente dalla base perché implementa il metodo virtuale puro della base.
  Contiene i campi dati privati aggiuntivi caratteristici di un film `plot, distribution, duration` rispettivamente trama, casa di produr./distribuzione e durata in ore:minuti del film.

3.1.2 Il contenitore

Container.h è una classe esterna alla gerarchia G che contiene la **definizione completa di un opportuno contenitore C**, con relativi **iteratori**, che permettono inserimenti, rimozioni e modifiche.

Il contenitore C è dal punto di vista progettuale un **template di classe** (perciò definito completamente in un unico file header) con due classi **Nodo** e **Smartp** annidate ed un unico campo dati `Smartp first` nella sua parte privata.

Smartp ha come unico campo dati un `Nodo* punt` mentre la classe Nodo ha 3 campi dati `T info; Smartp next; int riferimenti`.

Nella parte pubblica di Container è definita la classe annidata *Iterator* con un campo dati privato `Smartp punt` per accedere agli elementi nel contenitore.
Le classi *Container* ed *Iterator* sono infatti reciprocamente amiche.

Iterator **ridefinisce gli operatori** di incremento (post e prefisso), dereferenziazione, uguaglianza e disuguaglianza per modificare gli elementi, mentre *Container* ha i vari metodi di push e pop per manipolare la lista e begin(), end() e la ridefinizione dell'operatore di indicizzazione che usano Iterator.

3.1.3 Classe User

E' una classe concreta esterna a G che rappresenta l'entità utente per accedere all'applicazione.
I suoi campi dati sono: `username, password, name, surname` ed infine un c.d. `Container<const Media*> mediaDatabase;` che rappresenta il contenitore di tutti i media appartenenti allo *User*.

3.1.4 Classe Database

E' una classe concreta esterna a G che rappresenta il database degli utenti di Mediary ed ha infatti come suo unico campo dati privato il campo `Container<const User*> userDatabase;` .

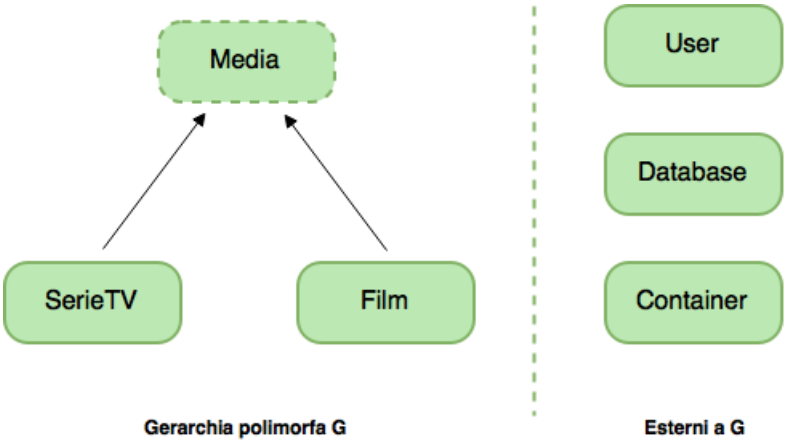

Figura 2: Entità che compongono il Model

3.2 View

L'intera *VIEW* è rappresentata da una **gerarchia polimorfa** che fa capo alla classe base astratta **MainView** . Essa deriva pubblicamente a sua volta dalla cl.astratta di Qt *QWidget* .

In *MainView* è definito il metodo *virtuale puro*

```
virtual void loadGraphic() =0;
```

che permette di **caricare la view POLIMORFA** corretta in base al tipo dinamico del puntatore della classe derivata.

3.2.1 Classi e gerarchia

Le classi derivate da *MainView* sono:

- **loginView**: E' la classe che si preoccupa di costruire la view di start all'avvio dell'applicazione.
- **userView**: E' la classe che costruisce la view per uno User. Ci si accede una volta eseguita correttamente l'autenticazione in loginView.
- **serietvView**: E' la classe che costruisce la view di una SerieTV. Viene creata a partire dalla *userView* quando si vuole creare o modificare una SerieTV.
- **filmView**: E' la classe che costruisce la view di un Film. Viene creata a partire dalla *userView* quando si vuole creare o modificare un Film.
- **userDataView**: E' la classe che costruisce la view per vedere/modificare i dati di uno User. Vi si accede sempre dalla *userView* quando si clicca nel bottone dedicato.
- **registrationView**: E' la classe che costruisce la view per la registrazione di un nuovo User. In alternativa al login cliccando nel bottone per la registrazione si accede ad essa.
- **dialogMessage**: E' una classe per la rappresentazione di una finestra che da una view di dialogo con l'utente, composta da un titolo, un messaggio ed un bottone che se cliccato è collegato alla chiusura della finestra stessa. *dialogMessage* deriva pubblicamente dalla classe di Qt *QDialog* .

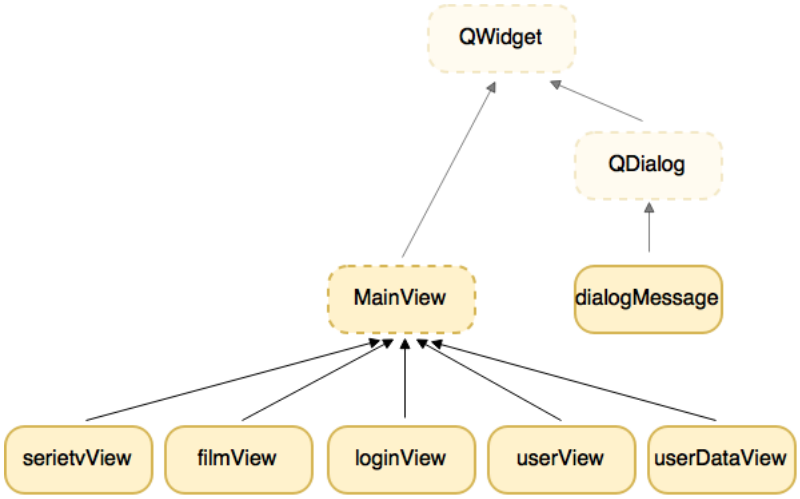

Figura 3: Gerarchia della View

3.2.2 Presentazione

All'avvio di Mediary si apre la view di login; in alto a destra è presente il bottone *esci* per chiudere il programma. Nel caso si eseguisse con successo la procedura di l'autenticazione cliccando nel login button si verrebbe mandati alla finestra personale con la possibilità di aggiungere, modificare oppure eliminare media.

Nel mezzo della *userView* è presente una tabella che permette di visualizzare, in ordine di inserimento dal più al meno recente, i media divisi per serieTv, film oppure tutti i tipi (l'opzione è selezionabile tramite una comboBox).

E' possibile modificare un elemento cliccando al di sopra **titolo** del media che si intende manipolare, mentre per cancellarlo è necessario cliccare sopra l'icona del cestino nell'ultima colonna della tabella corrispondente alla riga del media (si aprirà un avviso per la conferma).

La stessa finestra offre inoltre la possibilità di modificare i propri dati personali cliccando nel bottone "Gestisci dati utente".

Ogni finestra per la manipolazione dei dati del model ha i suoi controlli di consistenza dei dati.
Infine, nella parte più bassa della view utente c'è un bottone per il logout che riporta alla *loginView* con la possibilità di entrare in quella per la registrazione.

Figura 4: Start app

3.3 Controller

Il *CONTROLLER* è composto da sole due classi che però riescono a gestire ordinatamente tutte le funzioni e le view del programma:

3.3.1 mediaryController

Nel *main* del programma si crea subito un oggetto `controller` di tipo *mediaryController* che costruisce subito la *loginView*. Nel caso in cui l'utente effettui il login viene chiamata la funzione `verifyLogin(..)` che in caso positivo permette l'apertura della *userView* personale passando la gestione allo *userController*.

In caso differente chiama la funzione `openRegistrationView()` che costruisce la view di registrazione.

3.3.2 userController

Questa classe connette le restanti funzionalità che sono accessibili una volta entrati nella view personale, ovvero quelle di **creazione, modifica, cancellazione dei media, la gestione dei propri dati ed il logout.**

4 Database

La parte che compone il database dell'applicazione è stata realizzata in XML con l'utilizzo di due file: userDatabase.xml per l'archiviazione degli utenti e **nomeutente**mediaDatabase.xml per l'archiviazione dei media di ogni utente.
Come fa intuire il secondo file antepone un nome differente per ogni database dei media per uno specifico utente.

4.1 Gestione

A livello procedurale le classi che si occupano di gestire il database sono due, **Database e User**. Di seguito la descrizione del processo:

Si parte dalla classe **Database** che all'esecuzione dell'applicazione, costruendo un oggetto *Database* (a suo volta nel costruito dal controller iniziale *mediaryController*) chiama la funzione `loadUserDb()` che carica tutti gli utenti presenti nel file *userDatabase.xml* .
Una volta eseguito correttamente il login viene caricato il database dei media di quell'utente con la chiamata dello *userController* della funzione `loadMedia()` della classe **User** leggendo dal file *mediaDatabase.xml*.
Al contrario, alla chiusura della vista utente tramite logout viene chiamata la funzione di salvataggio dei media personali `writeMedia() const` che al suo interno esegue la chiamata **polimorfa** di `saveMedia(xmlWriter& )`, già citata precedentemente, chiamando correttamente quella adatta al tipo dinamico del puntatore al media nel container e scrivendo in *mediaDatabase.xml* i media aggiornati.
Infine alla chiusura dell'applicazione, la funzione `saveUserDb()` della classe *Database* chiama al suo interno la funzione di scrittura `writeUser(xmlWriter& )` della classe *User* per ogni utente, scrivendo cosi i aggiornati in *userDatabase.xml*

5 Estensibilità

Per quanto riguarda l'estensibilità e l'evolvibilità del programma, è disponibile ovviamente sia l'aggiunta orizzontale di altre classi derivate dalla base astratta *Media* (per esempio classe *Documentari* oppure *Sport*), sia quella verticale ampliando le classe derivate con sottoclassi più specifiche (come per esempio *Avventura o Fantascienza* dalla cl. Film) . Sarebbe inoltre possibile e sicuramente più incisivo dal punto di vista di espansione del progetto e delle sue funzionalità, aggiungere altre categorie di utenti (per esempio a pagamento e free ecc) creando una gerarchia polimorfa come per i media e posizionando come base astratta di esse la classe esistente *User*. Ciò estenderebbe le funzionalità per tipo di utente ed evolverebbe il codice rendendolo polimorfo in alcune sue parti, come per esempio il salvataggio degli utenti tramite la funzione già descritta `writeUser(xmlWriter& )`, come avviene già per i media.

```
for(Container<const User*>::Iterator it=userDatabase.begin(); it!=userDatabase.end(); ++it)
  userDatabase[it]->writeUser(xmlWriter); //PRONTO PER ESTENSIBILITA' POLIMORFA
```

6 Informazioni utili

6.1 Comincia subito

Per provare subito ad accedere tramite login è disponibile il seguente utente già registrato all'app:

- **username**: jacky
- **password**: jack93

6.2 Ambiente di sviluppo

Versione Qt	5.3.2
Ambiente di sviluppo	Qt Creator 3.2.1 (opensource)
Versione compilatore	Clang 5.0 (Apple), 64 bit
SO della macchina	OS X El Capitan versione 10.11.6