Relazione progetto Qt

Christian Libralato, MAT: 2101047

1 Introduzione

Il programma si articola sulla base del concetto di collezione personale di un utente, intesa come un insieme di oggetti fisici o virtuali posseduti, che può essere aggiornata aggiungendo, rimuovendo o modificando gli elementi. È possibile effettuare ricerca di singoli elementi o applicare filtri per la visualizzazione, inoltre vi è la possibilità di visualizzare i dettagli di un elemento e di riprodurlo, qualora faccia parte di una categoria compatibile.

Le categorie di oggetti della collezione sono fisse e consistono in Libro, Film, Canzone, Elemento qualunque. L'ultima permette di introdurre elementi di tipo diverso dai precedenti, ovviamente gli oggetti riproducibili sono di tipo Film o Canzone.

2 Descrizione Modello

Nel modello logico è contenuta la gerarchia di elementi che costituiscono la collezione e l'insieme di classi ausiliarie necessarie per implementare le funzionalità di persistenza dei dati , ricerca e filtraggio.

La gerarchia si basa sulla classe progenitrice *item* , classe base astratta polimorfa, che contiene le informazioni comuni ad ogni tipo di oggetto. Sono degni di nota i metodi virtuali:

- *isRelevant(filter : const filter&) : bool*: determina se l'oggetto di invocazione viene inserito nella lista di elementi risultante dall'applicazione di un filtro.
- *howRelevant(keyword : const QString&) : float*: utilizzato per la funzionlità di ricerca, ritorna un punteggio compreso tra 0 e 1 determinato dalla similarità tra la parola chiave inserita nella barra di ricerca e i campi dell'oggetto di invocazione.
- *accept(visitor : IVisitor&) : void*: metodo virtuale puro, implementa il design pattern Visitor per costruire widgets della GUI per ogni tipologia di elemento.
- *accept(jVisitor: jsonVisitor&) : void*: implementa il design pattern Visitor per la persistenza dei dati.

La classe *item* è anche virtuale in quanto da essa derivano virtualmente le classi astratte polimorfe *media* e *digital*. *media* racchiude gli articoli di matrice artistica, dunque opere letterarie, cinematografiche e musicali, mentre *digital* include gli articoli di natura virtuale/digitale e dunque riproducibili. Entrambe le classi presentano nuovi campi e metodi propri. Da esse derivano le classi concrete *book*, *movie*, *song* mentre la classe concreta *generalItem* deriva direttamente da *item*, ognuna di esse è dotata di campi e metodi propri.

La collezione è rappresentata tramite la classe *collection* ed implementata come un vettore di puntatori polimorfi ad *item*. La classe contiene metodi per la persistenza dei dati e per aggiungere o eliminare elementi, in particolare il metodo *applyFilter(const filter&) : QVector<item*>* genera una lista risultatante dall'applicazione di un filtro mentre *makeSearch(const QString& keyword) : QVector<item*>* genera la lista risultante da una ricerca.

Per motivi di leggibilità il diagramma UML del modello logico è fornito come file separato.

3 Persistenza dei dati

Il metodo implementato per la persistenza dei dati sfrutta il formato json e il design pattern del visitor. Infatti, dato che ogni classe necessita di operazioni dedicate per la sua traduzione in formato json, viene

definita la classe *jsonVisitor* dedicata al salvataggio e che implementa overloading dei metodi visit per ogni tipo di elemento e per il tipo *collection* costruendo un *QJsonObject*. Il *QJsonObject* ottenuto dalla visita su *collection* consiste nell'intera lista di elementi tradotta in formato json, da esso viene costruito il file .json finale nel metodo *collection::saveCollectionAs() : bool*. Per facilitare il processo di lettura, in aggiunta ai campi di ogni oggetto viene salvato un valore che ne definisce la tipologia

Per quanto concerne la lettura invece viene utilizzata la classe jsonReader che permette di leggere un'intera *collection* utilizzando, per lettura di singoli oggetti, il metodo *read(jObj : const QJsonObject&) : item**, sfruttando il campo che definisce il tipo viene costruito l'elemento corrispondente con i campi letti dal .json.

4 Funzionalità implementate

Le funzionalità implementate richieste dalla consegna del progetto consistono in:

- Gestire quattro tipologie di elemento
- Visualizzare tutti gli elementi in forma di lista o visualizzare i dettagli di particolare elemento.
- Creare, modificare, eliminare gli elementi e, più in generale, creare e modificare l'intera collezione.
- Salvare una collezione o aprirne una esistente.
- Effettuare ricerca di elementi.
- Navigare tra diverse schermate tramite interfaccia

Come funzionalità aggiuntive sono presenti:

- Applicazione di filtri alla visualizzazione altamente personalizzabili
- Riproduzione di articoli digitali
- Controllo modifiche non salvate prima della chiusura
- Minime scorciatoie da tastiera per i comandi della toolbar: save = Ctrl+S, open = Ctrl+O, newCollection = Ctrl+N
- Sidebar espandibile
- Utilizzo di icone nei pulsanti
- Personalizzazione resa estetica dell'interfaccia

È sicuramente utile definire l'idea su cui si basano le funzionalità di ricerca e filtraggio.
Nel file *utilityfunctions* sono contenute le funzioni:

- *QString longestCommonSubString(const QString& src,const QString& keyword)* : è una funzione che date due stringhe A e B restituisce la più lunga sottostringa comune tra le due.
  Data la stringa $A =< a_1, a_2, ..., a_n >$, una *sottostringa* di A è una stringa S tale che $S =< a_{i_1}, a_{i_2}, ..., a_{i_k} >$ dove $i_1 < i_2 < ... < i_k$ e gli indici i_t e i_{t+1} con $t \epsilon \{1, ..., k-1\}$ differiscono esattamente di 1.

  Per ogni elemento di *src* un secondo contatore aumenta finché vi è corrispondenza con gli elementi di *keyword* salvando la sottostringa corrente, il controllo di corrispondenza tra caratteri è case insensitive. Terminata la corrispondenza o raggiunta la fine viene effettuato un controllo sulla lunghezza per aggiornare la più lunga sottostringa ottenuta fino a quell'indice. Con questo metodo la ricerca sarà efficace anche con errori di battitura purché non vi siano lettere errate nelle prime posizioni della parole cercata.

- *float evaluateLCSS(const QString& src,const QString& keyword)*: permette di assegnare un punteggio compreso tra 0.0 e 1.0 direttamente proporzionale alla pertinenza di *keyword* in *src* confrontando la lunghezza della sottostringa comune ottenuta con la funzione precedente e la lunghezza di *keyword*.

- funzioni per l'Heapsort: si tratta di funzioni che implementano l'agoritmo di ordinamento heapsort con la modifica di avere due vettori come parametro, l'ordinamento viene effettuato secondo i valori del primo ma gli scambi si ripercuotono su entrambi.

La ricerca si articola seguendo i seguenti passaggi: ottenuta la *keyword* viene assegnato un punteggio di pertinenza ad ogni elemento della collezione tramite la funzione *evaluateLCSS* applicata ad ogni campo dati compatibile, in particolare i campi dati stessi hanno dei livelli di importanza nella ricerca (lowPriority, mediumPriority, highPriority) che ne influenzano la rilevanza, il livello di pertinenza di un elemento coincide con il massimo dei punteggi ottenuti dai campi dati, ciò avviene nei metodi *item::howRelevant(...)*.

Viene dunque costruita una lista di punteggi di pertinenza, all'interno del metodo *collection::makeSearch(...)*, a cui viene applicato l'heapsort assieme alla lista di indici degli elementi corrispondenti. Come risultato si ottiene la lista di indici della collezione, ordinati secondo la pertinenza, che viene utilizzata per costruire il widget di visualizzazione lista *showListWidget*

L'applicazione dei filtri avviene in maniera simile. Viene sfruttato l'oggetto *filter* che consiste in un metafiltro, contiene puntatori, nulli di default, a oggetti dello stesso tipo dei campi dati degli elementi, sono inclusi solo i tipi di campo più indicati per l'applicazione di un filtro.

A seguito della costruzione del filtro il metodo *item::isRelevant(...)* controlla se l'oggetto rispetta le condizioni del filtro, se vi è almeno un campo dati che non rientra nei valori target o vi è almeno un campo dati attivo (diverso dal puntatore nullo) del filtro che non è compatibile con il tipo di oggetto corrente, allora l'oggetto è considerato non pertinente; ad esempio, se un limite al numero di pagine è attivo nel filtro, tutti gli elementi di tipo diverso da *book* non vengono considerati.

5 Polimorfismo

Il principale utilizzo del polimorfismo riguarda le funzionalità di filtraggio, ricerca e la costruzione di widget mediata dall'*IVisitor*.

L'applicazione di filtri, allo stesso modo della risoluzione di una ricerca, richiede che gli oggetti effettuino dei controlli di rilevanza in base ai loro campi dati, per cui si ha utilizzo di polimorfismo per le implementazioni dei metodi *howRelevant(...)* e *isRelevant(...)*.

La creazione di un widget dipende oltre che dall'oggetto target anche dal tipo di widget richiesto, a ciò rispondono le classi che ereditano da *IVisitor* , che fanno ampio uso di polimorfismo, che sono *addEditVisitor*, *detailVisitor*, *previewVisitor* e *mediaplayerVisitor*, esse resituiscono rispettivamente widget di tipo *classeditor*, *classdetail*, *classpreview* e *classplayer*. Ognuno di questi widget ha un'implementazione per ogni tipo di elemento e i visitor, in base all'oggetto di invocazione, restituiscono quella corretta.

6 Struttura della GUI

L'interfaccia presenta una struttura fissa composta da:

- *toolbar*: permette salvataggio, apertura e creazione di nuova collezione.
- *sidebar*: offre gli strumenti per la ricerca e la creazione di filtri.
- un widget variabile che permette la visualizzazione a lista degli elementi (inclusi i risultati di ricerca e filtri), dettaglio, modifica ed eventuale riproduzione.

Le categorie di widget della parte variabile vengono costruiti e restituiti dai visitor sono:

- *classpreview*: compone l'anteprima di un oggetto, utilizzata per la visualizzazione a lista nello *showlistWidget*.
- *classeditor*: definisce il widget per la modifica, utlizzato anche per la creazione, il comportamento è definito da un flag.
- *classdetail*: visualizza i dettagli di un oggetto.
- *classplayer*: riproduce il contenuto multimediale se la classe è compatibile.

Ogni classe concreta possiede una propria implementazione di queste classi, alcuni esempi sono *bookpreview*, *moviepreview* etc., ovviamente per quanto riguarda *classplayer* esistono solo *movieplayer* e *songplayer*.

In molti casi per permettere la comunicazione tra widget distanti nella gerarchia si è ricorso a segnali di inoltro nelle classi intermedie per propagare verso l'alto un segnale proveniente da un widget più interno. L'interfaccia è infine stata arricchita con icone e colori personalizzati tramite stylesheets.

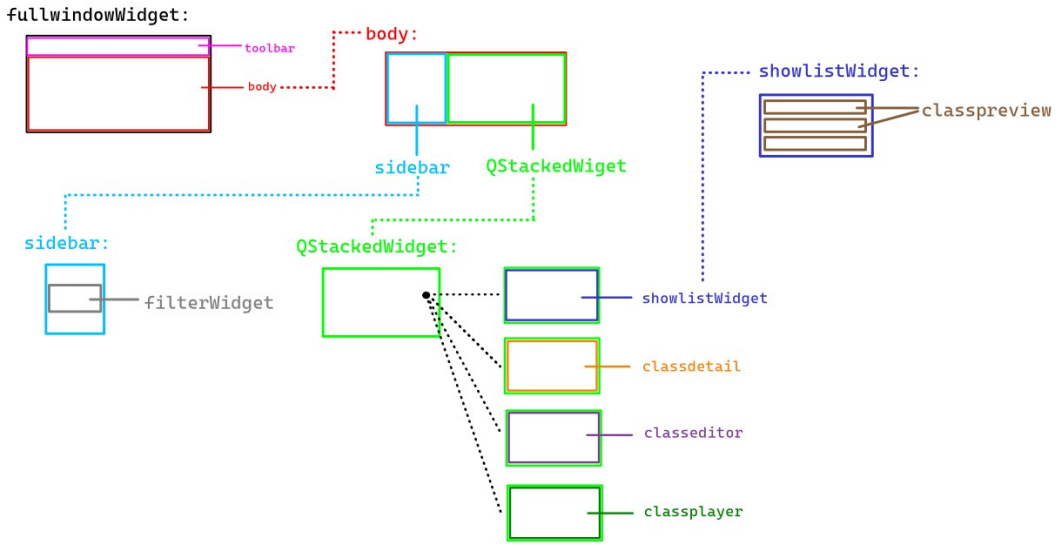

Figure 1: Rappresentazione intuitiva della struttura della GUI

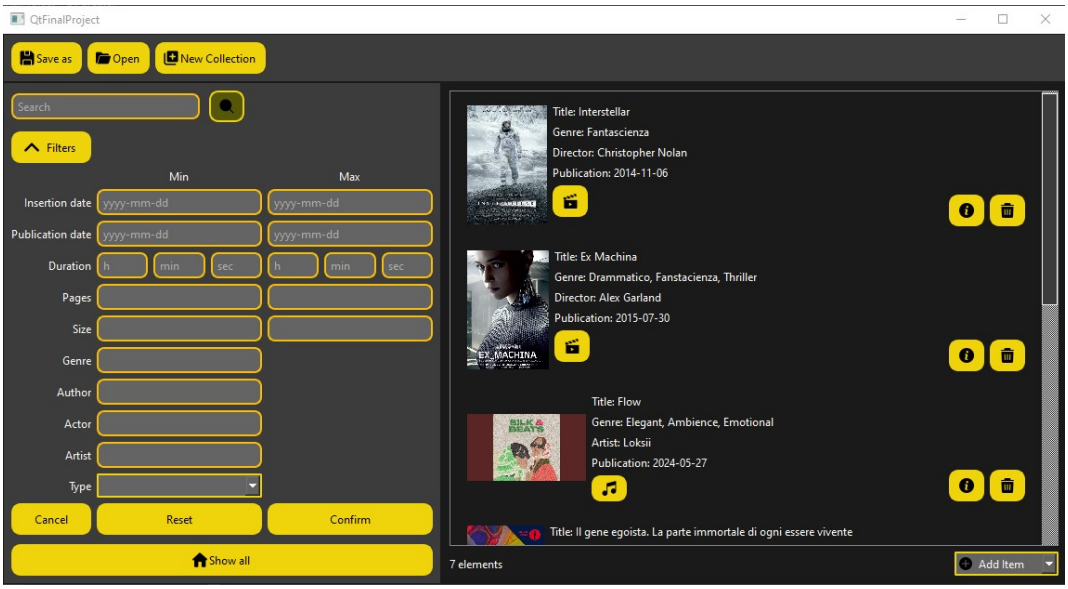

Figure 2: Schermata principale

7 Tempi di realizzazione

Il monte ore previsto è stato ampiamente superato in parte per le funzionalità aggiuntive e in parte a causa della GUI, in particolare per l'implementazione corretta dei widget restituiti dai visitor e per il metodo di comunicazione tra finestre non direttamente legate da relazione padre-figlio, inoltre è incluso

il tempo di realizzazione della resa grafica personalizzata. Ulteriore contribuzione alla dilatazione dei tempi è stata la fase di testing a causa di problemi di compatibilità con il player e la macchina virtuale moodle.

Attività	**Ore**
Progettazione	12
Sviluppo codice modello	28
Sviluppo codice GUI	39
Test e Debug	10
Scrittura relazione	4
Totale	93