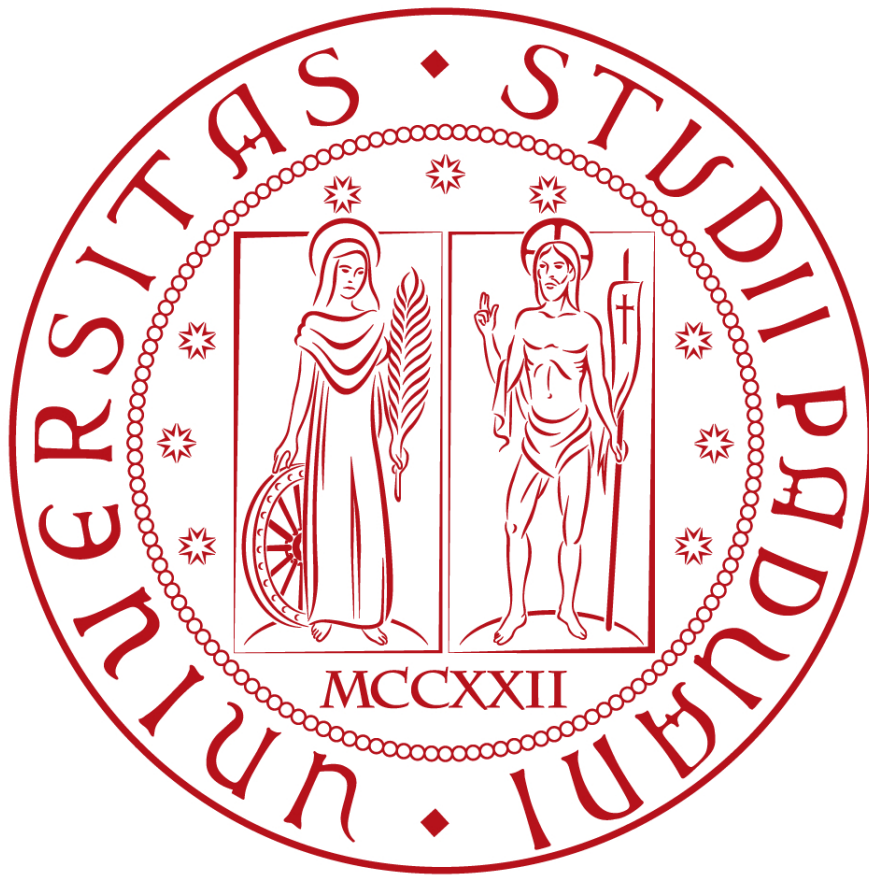


Relazione Progetto di Programmazione ad oggetti



Email: pietro.gabelli@studenti.unipd.it

Indice

1	Introduzione	2
1.1	Scopo del documento	2
1.2	Scopo del progetto	2
1.3	Specifiche progettuali	2
2	Classi logiche	3
2.1	Gerarchia oggetti memorizzati	3
2.2	Classe contenitore	4
2.2.1	ContainerItem	4
2.2.2	Iterator	4
2.2.3	const_Iterator	5
2.3	Classe SmartPtr	5
2.4	Classe Biblio	6
3	Classi Grafiche	7
3.1	QWidget	7
4	Gestione della memoria	8

Elenco delle figure

1	Model/View architecture	3
2	Gerarchia delle classi	3
3	Operazioni previste	7

1 Introduzione

1.1 Scopo del documento

Lo scopo di questo documento è di presentare in maniera chiara le scelte architetturali fatte.

1.2 Scopo del progetto

Il progetto ha come scopo lo sviluppo di un'applicazione per la gestione di una biblioteca. Gli oggetti che compongono la biblioteca sono inseriti in un database che viene utilizzato attraverso una interfaccia grafica. Lo sviluppo è stato fatto utilizzando C++ e Qt.

1.3 Specifiche progettuali

Il progetto è destinato ad immagazzinare i titoli di una biblioteca domestica. Gli oggetti che si possono salvare sono:

- Libri;
- Film, siano questi VHS o DVD.
- CD;

Le funzionalità principali che si intende modellare sono:

- Aggiunta di nuovi elementi;
- Ricerca tra i titoli presenti;
- Cancellazione di titoli, tra quelli presenti;
- Persistenza su disco dei dati inseriti.

I vincoli di progetto sono i seguenti:

1. Definizione ed utilizzo di una gerarchia G di tipi di altezza ≥ 1 e larghezza ≥ 1 .
2. Definizione di un opportuno contenitore C , con relativi iteratori, che permetta inserimenti, rimozioni, modifiche.
3. Utilizzo del contenitore C per memorizzare oggetti polimorfi della gerarchia G .
4. Il front-end dell'applicazione deve essere una GUI sviluppata nel framework Qt.

Il progetto è stato sviluppato in un sistema XUbuntu; è stato utilizzato l'IDE Qt Creator nella versione 3.0.1, con le librerie alla versione 5.2.1.

Si è provato il progetto sui computer del Laboratorio Informatico Plesso Paolotti (LabP140 - labP036) dove compila ed esegue correttamente.

Il database che compone l'applicazione viene aperto e salvato usando il formato XML: entrambe le operazioni avvengono in maniera automatica grazie ad un path impostato di default.

Nello sviluppo è stata curata la separazione tra la parte logica e la parte grafica, senza utilizzare il design pattern MVC. Si è preferita invece un'architettura Model/View, che rende possibile la separazione del modo in cui sono immagazzinati i dati e la loro presentazione all'utente.

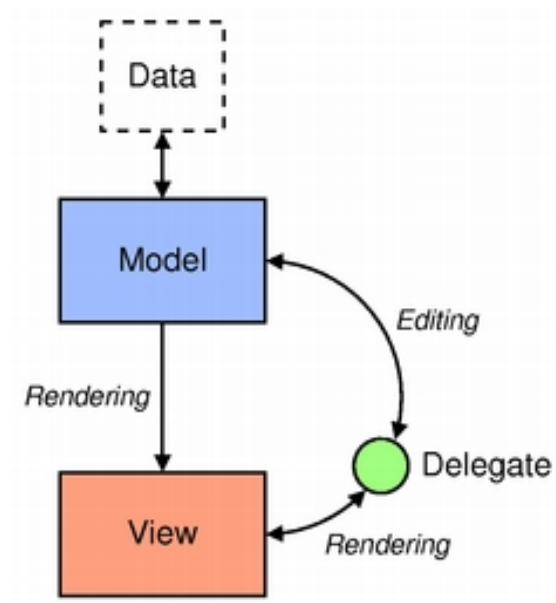


Figura 1: Model/View architecture

2 Classi logiche

2.1 Gerarchia oggetti memorizzati

La gerarchia sviluppata è composta da 5 classi:

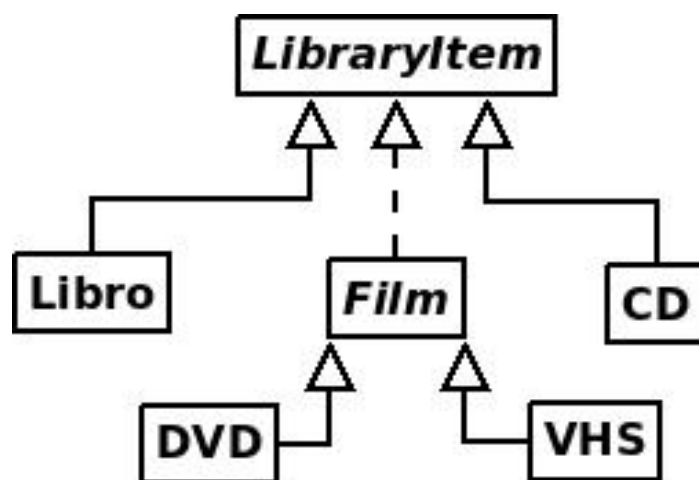


Figura 2: Gerarchia delle classi

La classe **LibraryItem** rappresenta un oggetto generico inserito nella libreria; per questo motivo ho scelto di renderla polimorfa ed astratta: non è possibile dichiarare oggetti di questa classe. All'interno di questa classe sono memorizzati gli attributi comuni a tutti gli oggetti della gerarchia: *titolo*, *genere*, entrambi di tipo string. Per questa classe e per le altre classi nella gerarchia verranno resi disponibili:

- Un costruttore di default ed un costruttore per i campi dati della classe;
- Un distruttore virtuale;
- Un metodo *clone* che restituisce il puntatore ad una copia all'oggetto su cui viene invocato;
- Un operatore di uguaglianza ed uno di disuguaglianza;
- Metodi **get** che restituiscono una copia dei campi dati, per ogni campo dati richiesto.

Le classi **Libro**, **CD**, **Film** sono classi derivate direttamente dalla classe base **Utente**; sono tutte concrete, a differenza di **Film**, polimorfa ed astratta come **LibraryItem**.

Le classi **DVD**, **VHS** sono classi concrete derivate da **Film**.

Per quanto riguarda i campi dati:

- La classe **Libro** contiene:
 - Un *autore*, memorizzato tramite uno string;
 - Un *anno di uscita*, memorizzato tramite un intero;
 - Un *editore*, memorizzato tramite uno string;
 - La definizione dei metodi viruali definiti nella classe **LibraryItem**.
- La classe **Film** contiene:
 - Un *regista*, memorizzato tramite uno string;
 - Una *durata* in minuti, memorizzata tramite un intero;
 - Una *data d'uscita*, memorizzata tramite un `QDate`;
 - La definizione dei metodi viruali definiti nella classe **LibraryItem**.
- La classe **CD** contiene:
 - Un *artista*, memorizzato tramite uno string;
 - Un *anno di uscita*, memorizzato tramite un intero;
 - Un *numero di dischi*, memorizzato tramite un intero;
 - La definizione dei metodi viruali definiti nella classe **LibraryItem**.
- Le classi **CD**, **DVD** sono implementazioni di **Film**; non contengono ulteriori campi dati. Rendono disponibili costruttori, un distruttore, il metodo clone, operator `==` e `!=` che richiamano quelli della superclasse **Film**.

2.2 Classe contenitore

Per memorizzare gli oggetti della biblioteca è stato creato il template di classe **Contenitore**.

La classe è dichiarata amica di *operator*», delle classi **iterator** e *const_iterator*; all'interno è dichiarata privata la classe **ContainerItem**; sono invece pubbliche le classi **Iterator**, **const_iterator**.

2.2.1 ContainerItem

La classe è caratterizzata da:

- l'oggetto *info*;
- puntatore all'oggetto **ContainerItem** successivo;
- il costruttore di default ed il costruttore a 2 parametri;
- operatore d'uguaglianza e disuguaglianza;

Essendo la classe racchiusa nella parte privata, metodi e campi dati sono stati dichiarati pubblici.

2.2.2 Iterator

La classe è caratterizzata da:

- la dichiarazione di amicizia con la classe **Container**
- un puntatore a **ContainerItem**, dichiarato privato;
- costruttore di default;
- operatore d'uguaglianza e disuguaglianza;
- operatori d'indirizzione ed indicizzazione;
- operator `++` prefisso, postfisso;
- operator* che restituisce copia dell'oggetto puntato.

2.2.3 `const_iterator`

La classe è caratterizzata da:

- la dichiarazione di amicizia con la classe **Container**
- un puntatore a **const ContainerItem**, dichiarato privato;
- costruttore di default;
- convertitore di tipo da **Iterator**;
- operatore d'uguaglianza e disuguaglianza;
- operatore `++` prefisso, postfisso;
- operatore di dereferenziazione per ottenere un riferimento costante all'oggetto puntato, marcato costante.

Metodi di **Container**

Sono dichiarati privati:

- puntatore al primo elemento **ContainerItem**;
- un metodo dichiarato statico *deepcopy* che fa una copia profonda della lista;
- un metodo dichiarato statico *deepremove* che esegue la distruzione profonda della lista;
- un metodo dichiarato statico *equals* che, dati due puntatori a **ContainerItem**, l'uguaglianza delle liste puntate.
- un metodo dichiarato statico *printcontainer* per effettuare la stampa su ostream.

Sono pubblici:

- un costruttore di default;
- il costruttore di copia (profonda);
- l'operatore d'assegnazione (profonda);
- overloading del distruttore di default per fare la distruzione profonda;
- metodo *isEmpty* che dice se il contenitore è vuoto;
- metodo *insert* per inserire in testa un nuovo elemento nel contenitore;
- metodo *remove* per rimuovere un elemento;
- metodo *size*, restituisce il numero d'elementi presenti nel contenitore;
- metodo *replace* per sostituire un elemento con un altro;
- operatori d'uguaglianza, disuguaglianza;
- metodo *search* che restituisce un nuovo contenitore dove sono memorizzati gli elementi che corrispondono alla ricerca;
- metodi *begin*, *end*, che rispettivamente ritornano l'iteratore al primo nodo della lista ed un iteratore che punta ad un nodo vuoto (come quando si sta scorrendo la lista e non si è ancora arrivati al past-the-end); sono disponibili metodi equivalenti per ritornare gli equivalenti iteratori costanti;
- l'operatore di dereferenziazione che restituisce l'oggetto puntato da un iteratore; disponibile anche per l'iteratore costante, viene restituito un riferimento costante;

2.3 Classe **SmartPtr**

È stata definita una classe di puntatori smart, **SmartPtr**, che ha come campo dati privato un puntatore ad un oggetto della superclasse **LibraryItem**.

I metodi che sono stati resi disponibili per questa classe sono i metodi d'utilità per una classe di puntatori: *costruttore di default e ad un argomento*, *costruttore di copia*, overloading degli operatori di: *assegnazione*, *uguaglianza*, *disuguaglianza*, *indicizzazione*, *indirizzione*; *distruttore di default*. Si è infine definito un metodo *getItem* che restituisce copia dell'oggetto puntato; ne viene definito anche uno segnato costante.

2.4 Classe Biblio

Questa classe serve a concretizzare la classe templetizzata **Container** con le classi della gerarchia radicata in **LibraryItem** utilizzando **SmartPtr**, rendendo disponibili metodi di utilità che serviranno all'applicazione. Contiene:

- Un **Container** di puntatori a **SmartPtr**;
- Un *distruttore*;
- Un metodo *isEmpty* che dice se non sono presenti elementi nella biblioteca;
- Un metodo *addItem*, che aggiunge un nuovo elemento alla biblioteca. Restituisce un booleano che indica la presenza di un'uguale opera all'interno della libreria. Sono permesse più copie d'uno stesso elemento.
- Un metodo *removeItem* per togliere un elemento dalla biblioteca;
- Un metodo *getItem* che data una posizione *i*, restituisce un puntatore all'*i*-esimo elemento nel **Container**;
- Il metodo *save* che su file il contenuto della biblioteca, in formato **XML**;
- Il metodo *load* che carica dal file XML il contenuto della biblioteca. Entrambi i metodi segnalano con un messaggio d'errore se non è stato possibile aprire il file.

3 Classi Grafiche

La parte View che si è modellata intende permettere all'utente di:



Figura 3: Operazioni previste

Per questo motivo, è stata creata una classe **QWidget**.

3.1 QWidget

Un oggetto di **QWidget** viene creato dal main senza parent, costruendo così un'*independent window*.

Questa classe funge da layout manager, contiene 5 groupbox al cui interno sono contenute le principali operazioni e funzionalità disponibili, divise nel modo seguente:

- *qgb_menu* rappresenta la groupbox in cui sono racchiusi i pulsanti delle operazioni sulla base di dati della biblioteca;
- *qgb_CD*, *qgb_Libri*, *qgb_DVD*, *qgb_VHS*, in cui sono racchiuse le tabelle che rappresentano - rispettivamente - i CD, Libri, DVD e VHS presenti nella biblioteca;

Ho scelto di usare le groupboxes per la possibilità di inserire un frame ed un titolo attorno alle tabelle. La creazione di una **MyWidget** provoca l'esecuzione de:

- L'impostazione del titolo della finestra, il ridimensionamento della stessa a seconda delle dimensioni dello schermo, lo spostamento al centro dello schermo; il tutto ad opera della funzione *setWidget()*;
- La creazione della **QGroupBox** *Qgb_Menu* ad opera della funzione *createMenuGroupBox()*, che connette anche i segnali dei pulsanti ai relativi slot.
- La creazione della **QGroupBox** *Qgb_CD* ad opera della funzione *createCdGroupBox()*;
- La creazione della **QGroupBox** *Qgb_Libri* ad opera della funzione *createLibriGroupBox()*;
- La creazione della **QGroupBox** *Qgb_DVD* ad opera della funzione *createDvdGroupBox()*;
- La creazione della **QGroupBox** *Qgb_VHS* ad opera della funzione *createVhsGroupBox()*.

Le quattro funzioni che creano le tabelle sono strutturate allo stesso modo, per semplicità verrà illustrata solamente *createCdGroupBox()*:

1. Il puntatore *qgb_CD* viene associato ad una nuova **QGroupBox** cui è stato assegnato il titolo: Catalogo CD;
2. Viene creata la tabella *tableWidget_CD* ed impostato il relativo header, composto dai nomi degli attributi che la compongono;
3. Viene creato un nuovo **QVBoxLayout**, cui viene aggiunta la widget associata alla tabella creata al punto 2;
4. Il layout della **QGroupBox** viene impostato a quello creato in precedenza.

La funzione *createMenuGroupBox()* provoca le seguenti azioni:

1. Il puntatore *qgb_Menu* viene associato ad una nuova **QGroupBox** cui è stato assegnato il titolo: Operazioni;
2. Vengono creati i pulsanti *Gestione CD*, *Gestione DVD*, *Gestione Libri*, *Gestione VHS*, *Trova Elemento*, *Chiudi* che servono ad aprire le finestre di gestione dei contenuti della libreria e permettono di chiudere il programma.

3. Viene creato *gridLayout*, dove verranno inseriti i pulsanti che compongono il menù; a questo vengono aggiunti i pulsanti creati in precedenza.
4. Vengono associati ai pulsanti i relativi slot; verranno illustrati solamente *slotFindItemQDialog()* e *slotInsertCD()*, dato che sono identici quelli di Libro, DVD e VHS.

Lo slot *slotInsertCD()* crea un nuovo oggetto della classe **Gestione_CD**; questo ha a disposizione una form dove inserire tutti i dati del CD per poi procedere con le 3 operazioni (inserimento, rimozione, cancellazione); sono disponibili anche segnali e slot che sono collegati con la QWidget principale che si occupa di invocare le modifiche sull'oggetto **Biblio**.

Nel caso dell'inserimento, la pressione del pulsante **Inserisci**, fa emettere il segnale *signalInsertCD()*, collegato allo slot *slotInsertCD()*; questo crea uno **SmartPtr** con l'oggetto restituito dallo slot *slotNewCD* della classe **GestioneCD** che prende campo per campo i valori inseriti nella form, istanza un nuovo oggetto CD e lo ritorna. A questo punto viene invocato il metodo *Insert* e *save* sull'oggetto **Biblio** e vengono aggiornate le tabelle nella QWidget.

I metodi di *slotRemoveCD()*, *slotRemoveCD()*, si differenziano per il messaggio d'errore che viene emesso nel caso non sia presente un CD con i campi dichiarati.

Lo slot *slotFindItemQDialog()* crea un nuovo oggetto della classe **TrovaElemento**, a cui viene passato l'oggetto **Biblio**. Ciò provoca l'apertura di una nuova QDialog, in cui si trovano i campi dati su cui è possibile effettuare la ricerca: titolo e genere (presenti in tutti i tipi), ed un campo autore, utilizzato al posto di regista ed artista per le classi **Libro** e quelle discendenti da **Film**; sono presenti anche delle checkBox per includere i campi nella ricerca, dei radioButton per precisare il tipo dell'oggetto da cercare ed una QTextEdit in cui sono inseriti i risultati.

Il pulsante che avvia la ricerca provoca l'avvio dello slot *slot_start_search()* provoca la cancellazione dei risultati della ricerca precedente, cui segue la ricerca all'interno degli elementi memorizzati nella biblioteca: viene individuato il tipo dell'oggetto corrente tramite *dynamic_cast* a cascata, qualora il tipo sia stato incluso nella ricerca, vengono prelevati i valori inseriti nei campi nella form e viene eseguito il confronto con l'oggetto corrente. Se la ricerca ha esito positivo, vengono inseriti su una QString tutti i campi dell'oggetto, nell'ordine inverso a quello che verrà mostrato usando il metodo *prepend*; tale QString viene poi appesa alla *QtextEdit* dei risultati, che viene aggiornata subito dopo.

4 Gestione della memoria

Nel progetto **Qbiblio** gli oggetti sono stati memorizzati nello heap.

La classe contenitore è stata dotata d'un distruttore profondo, che permette la deallocazione degli oggetti nello heap al momento della distruzione del puntatore.

Al momento della chiusura delle finestre il contenitore viene svuotato nel modo seguente: prima vengono deallocati gli oggetti puntati dagli **SmartPtr** e poi rimuove gli elementi del contenitore.