

# Qontainer

Elisabetta Piombin 1142189

## Indice

<b>1</b>	<b>Abstract</b>	<b>2</b>
<b>2</b>	<b>Manuale utente</b>	<b>2</b>
2.1	Finestra principale . . . . .	3
2.2	Aggiunta di contenuto . . . . .	4
2.3	Ricerca del Contenuto . . . . .	6
<b>3</b>	<b>Compilazione ed esecuzione</b>	<b>9</b>
<b>4</b>	<b>Descrizione della gerarchia di classi</b>	<b>9</b>
<b>5</b>	<b>Descrizione delle chiamate polimorfe</b>	<b>10</b>
<b>6</b>	<b>Descrizione del formato del file di salvataggio e caricamento del contenitore</b>	<b>10</b>
<b>7</b>	<b>Tempistiche</b>	<b>11</b>

## 1 Abstract

**Qontainer** è un progetto realizzato al fine di fornire un contenitore che gestisca una libreria di contenuti multimediali: file audio e file video, che si dividono a loro volta in canzoni, podcast, serie tv e film.

Per farlo, la classe templattizzata **container** a sua volta fa uso di altre classi annidate:

1. **nodo**: inserita nella parte privata di **container**, viene usata per memorizzare i vari contenuti multimediali, visti come se fossero una lista concatenata di elementi, con ogni nodo diviso nel suo campo **info** (di tipo parametrico T, **next** e **prev** (di tipo **nodo\***). In **container** è presente un puntatore al primo elemento della lista.
2. **const\_iterator**: inserita nella parte pubblica di **container**, è la classe che permette l'implementazione di iteratori costanti.
3. **iterator**: inserita nella parte pubblica di **container**, è la classe che permette l'implementazione di iteratori non costanti.

Vista l'assenza di puntatori ad altre classi nella gerarchia, è stata ritenuta superflua l'implementazione di una classe per un eventuale *smart pointer*.

Ho scelto di implementare il contenitore come una lista doppiamente linkata composta da nodi perché offro la possibilità di rimuovere contenuti all'interno della libreria in posizioni arbitrarie, e questo mi porta alla necessità di scegliere l'implementazione sotto forma di lista, che rende tale operazione di rimozione più efficiente (viene operata in tempo costante) rispetto alla scelta di implementare tale contenitore come un vettore di elementi.

## 2 Manuale utente

In questa sezione vengono descritte le varie schermate della GUI.

## 2.1 Finestra principale



Figura 1: Finestra principale del progetto

Da qui si ha un'*overview* dell'intera applicazione, con tutti i contenuti in essa inseriti dall'utente, suddivisi per **Canzoni**, **Podcast**, **Film** e **Serie**. Tramite la finestra principale si dà all'utente la possibilità di cercare o di aggiungere contenuto.

## 2.2 Aggiunta di contenuto

Qontainer - Elisabetta Piombin

# Aggiungi Contenuti

Scegli il tipo di file che vuoi aggiungere: Scegli: ▾

Titolo:

Durata:

Genere:

Dimensione:

Valutazione:

Autore:

Anno di uscita:

Aggiungi

Torna alla schermata principale

Figura 2: Finestra base di aggiunta contenuti

Qontainer - Elisabetta Piombin

## Aggiungi Contenuti

Scegli il tipo di file che vuoi aggiungere: Podcast ▾

Titolo:

Durata:

Genere:

Dimensione:

Valutazione:

Autore:

Anno di uscita:

Bitrate:

Raccolta:

Ospite:

Aggiungi

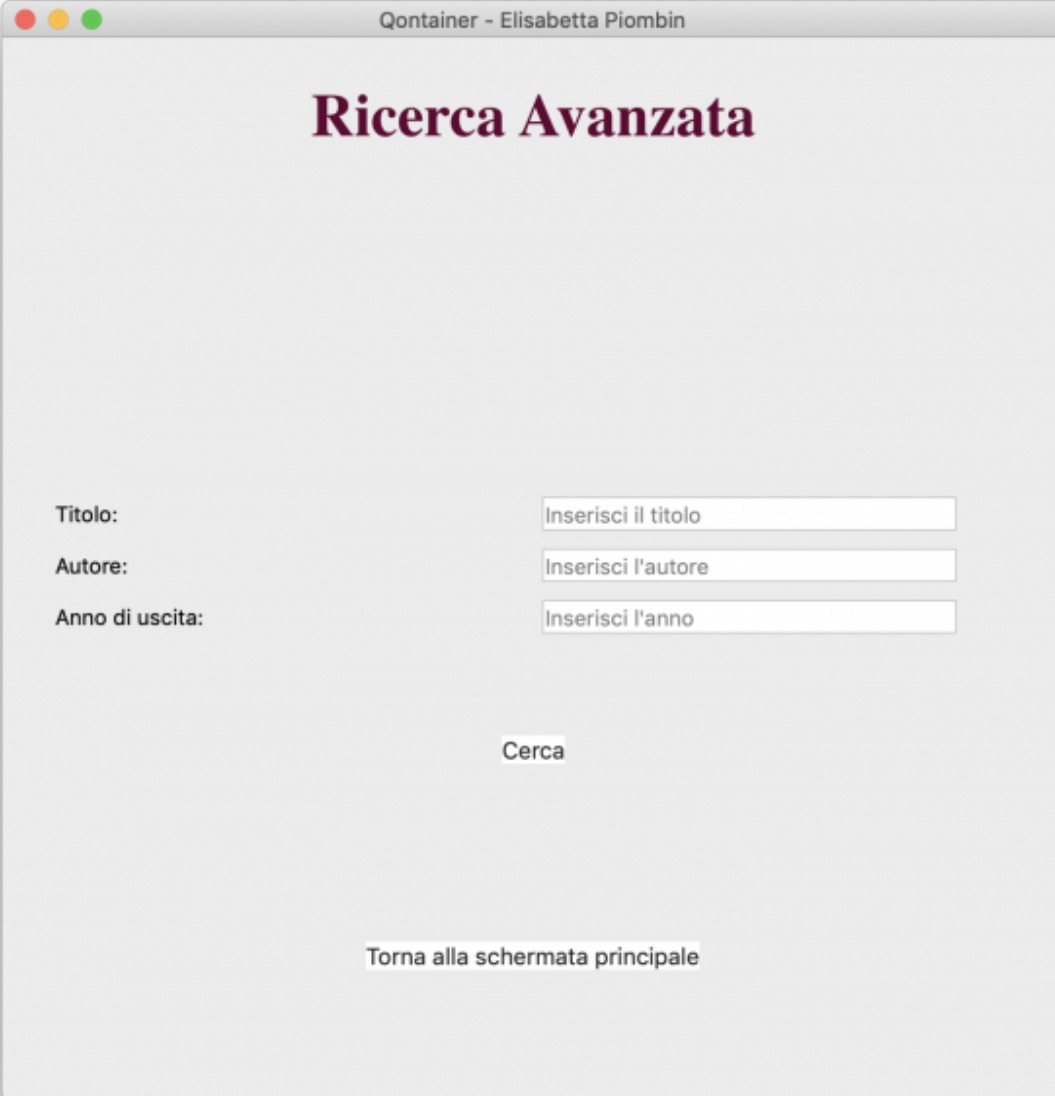
Torna alla schermata principale

Figura 3: Finestra specifica di aggiunta contenuti

Nella schermata di aggiunta dei contenuti l'utente deve scegliere il tipo di contenuto che vuole inserire, e a seconda della scelta compariranno i campi dati specifici di quella classe. Ad esempio, nella *Figura 3* è stato scelto di inserire un podcast, e questo ha fatto comparire i campi *Bitrate*, *Raccolta* e *Ospite*.

Dopo aver schiacciato il bottone *Aggiungi*, l'utente si trova una finestra di conferma di aggiunta riuscita.

## 2.3 Ricerca del Contenuto



Qontainer - Elisabetta Piombin

# Ricerca Avanzata

Titolo:

Autore:

Anno di uscita:

Cerca

Torna alla schermata principale

Figura 4: Prima finestra di ricerca

All'utente viene data la possibilità di modificare i contenuti esistenti, e la ricerca per la modifica è implementata in modo che accetti sia stringhe incomplete che *upper* o *lower case*. All'utente viene data solamente la possibilità di inserire i campi dati che è più probabile vengano ricordati a memoria, per avere una ricerca realistica: sarebbe impossibile che l'utente si ricordi il valore di ogni singolo campo dato di ogni oggetto inserito nell'applicazione.

Una volta inserito anche solo un campo dato parziale (nell'esempio ho inserito solamente "iron" nel campo del titolo), l'utente si vede restituire tutti i match trovati all'interno dell'applicazione:



Figura 5: Risultato della ricerca

Una volta aperta la finestra dei risultati, all'utente viene data la possibilità di **visualizzare** nel dettaglio, **modificare** oppure **eliminare** del contenuto.

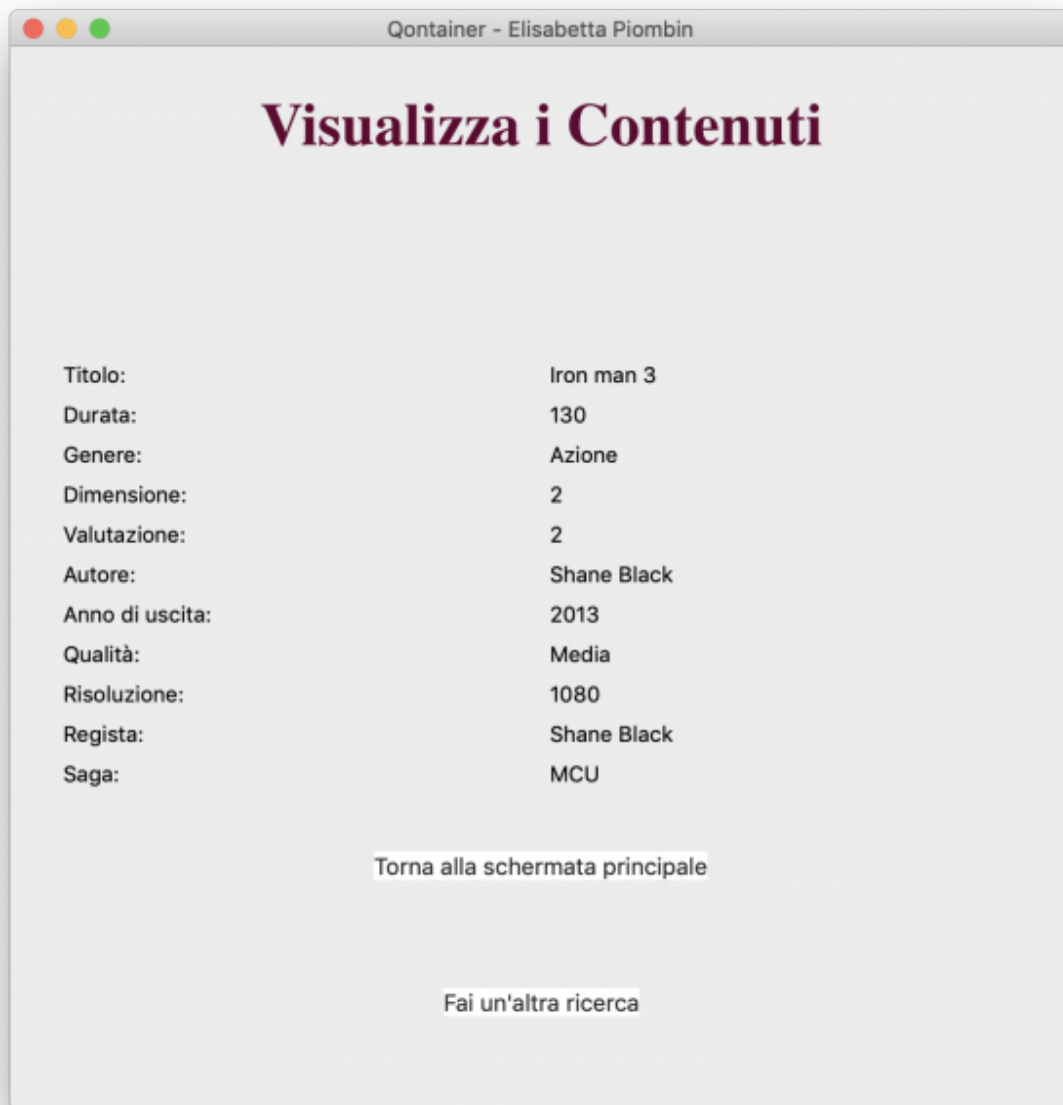


Figura 6: Visualizzazione dettaglio contenuti



Qontainer - Elisabetta Piombin

## Modifica Contenuti

Titolo:	Iron man 3
Durata:	130
Genere:	Azione
Dimensione:	2
Valutazione:	2
Autore:	Shane Black
Anno di uscita:	2013
Risoluzione:	1080
Regista:	Shane Black
Saga:	MCU

Conferma la modifica

Torna alla schermata principale

Figura 7: Modifica dei contenuti

### 3 Compilazione ed esecuzione

La compilazione si esegue tramite i comandi `qmake` e in seguito `make`.

### 4 Descrizione della gerarchia di classi

La classe base astratta da cui deriva tutta la gerarchia è `ContenutoMultimediale`, che verrà poi concretizzata tramite sue classi derivate. Da `ContenutoMultimediale` derivano immediatamente altre due classi: `audio` e `video`, che sono le due macrocategorie di appartenenza dei file che vengono memorizzati nella libreria.

Da `video` derivano due classi: `film` e `episodio`, mentre da `audio` derivano altre due classi, `podcast` e `canzone`.  
Non si verifica la situazione di ereditarietà multipla.

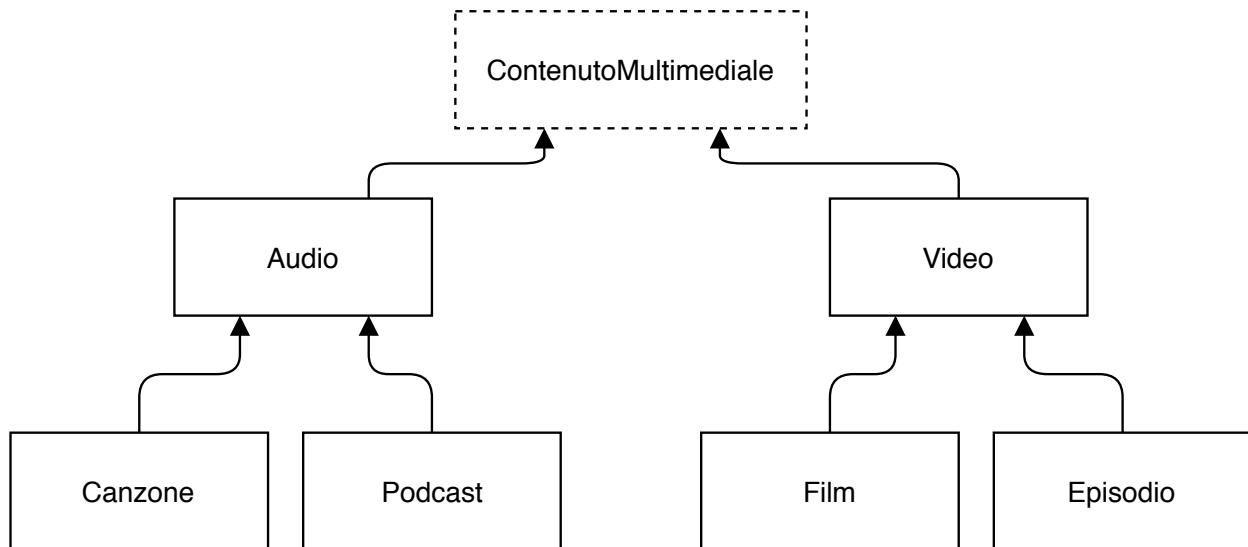


Figura 8: Gerarchia delle classi

## 5 Descrizione delle chiamate polimorfe

Sono presenti tre metodi virtuali nella gerarchia: `serializza(char)`, l'overloading dell'operatore di confronto `operator==(const T&)`, e il metodo `qualita()`, che è puro.

- `string serializza(char)`: definito virtuale all'interno di `ContenutoMultimediale`, viene implementato diversamente per i campi dati specifici di ogni classe derivata. Viene richiamata all'interno della classe `database`, all'interno della funzione `void save(container<T>&)`, in quanto questa è la funzione che si occupa del salvataggio su file degli oggetti inseriti dentro al `container`.
- `bool operator==(const T&)` è usato nella funzione `void remove(T)` all'interno di `ContenutoMultimediale`, e viene usata in `risultatoricerca` (nella GUI) per rimuovere un elemento dal contenitore. L'operatore di uguaglianza non viene usato nell'operazione di ricerca, nonostante potrebbe sembrare immediato il suo utilizzo in tale contesto. Per la ricerca dei contenuti all'interno del `container` ho implementato una funzione `vector<T> search (T)`.
- `string qualita() const`, metodo virtuale puro in `ContenutoMultimediale`, utilizzato in `visualizzaelemento` per dare la possibilità all'utente per vedere se il file che si sta visualizzando è di qualità media, alta o bassa. Ha diverse implementazioni in `audio` e in `video`.

Nella classe base `ContenutoMultimediale` è presente il distruttore virtuale di default.

## 6 Descrizione del formato del file di salvataggio e caricamento del contenitore

Per il caricamento dei file che compongono la libreria, e per il loro salvataggio, mi appoggio ad una classe esterna alla gerarchia, che ho chiamato `database`. Questa classe contiene i metodi `load()` e `save()`. La classe `database` non è un template perché dipende strettamente da `ContenutoMultimediale`.

- `void load(container<ContenutoMultimediale*>&)`, tramite un'operazione di deserializzazione implementata ad hoc in ogni classe derivata concreta della gerarchia, permette la lettura del contenuto in ogni file `.txt`. Esiste un unico file `.txt`, in cui sono scritti tutti gli oggetti che verranno caricati nel `container`. Database si occupa di caricare questi oggetti nella libreria tramite:

- (a) `static film* film::deserializza(const vector<string>&)`
- (b) `static episodio* episodio::deserializza(const vector<string>&)`
- (c) `static podcast* podcast::deserializza(const vector<string>&)`
- (d) `static canzone* canzone::deserializza(const vector<string>&)`

In modo che possano essere gestiti dal contenitore templetizzato.

- `void save(container<ContenutoMultimediale*>&)` si occupa di memorizzare sul file `.txt` i caratteri corrispondenti agli oggetti contenuti in `container`, invocando polimorficamente il metodo `serializza(char)` corretto, che è definito come virtuale.

## 7 Tempistiche

Ore	Fase
2	Analisi preliminare del problema
1	Progettazione del modello
1	Progettazione della GUI
10	Apprendimento libreria Qt
30	Codifica modello e GUI
10	Debugging
5	Testing