Relazione del progetto di Programmazione ad Oggetti: Qontainer

Elisabetta Piombin 1142189

Indice

1 Abstract

Qontainer è un progetto realizzzato 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 templatizzata `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` e `next` (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 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 Compilazione ed esecuzione

3 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. I metodi virtuali sono `riproduci()`, `pausa()`, `operator==` e il relativo distruttore; `riproduci()` e `pausa()` sono anche puri, poiché la loro implementazione è assegnata alle 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 ereditarità multipla.

4 Descrizione delle chiamate polimorfe

Le chiamate polimorfe vengono pesantemente utilizzate nelle operazioni di modifica, rimozione, ricerca e aggiunta degli elementi nel contenitore, in quanto il contenitore viene istanziato a `<ContenutoMultimediale*>`, e tramite *downcasting* (via `dynamic_cast`) effettuato al Tipo desiderato, si eseguono le operazioni necessarie sui tipi.
Le chiamate polimorfe presenti nel progetto sono le seguenti:

- nella *mainwindow* dell'applicazione, dove vengono elencati tutti i contenuti presenti nel contenitore: questa operazione è eseguita tramite downcasting da `ContenutoMultimediale*` al tipo target desiderato alla base della gerarchia.

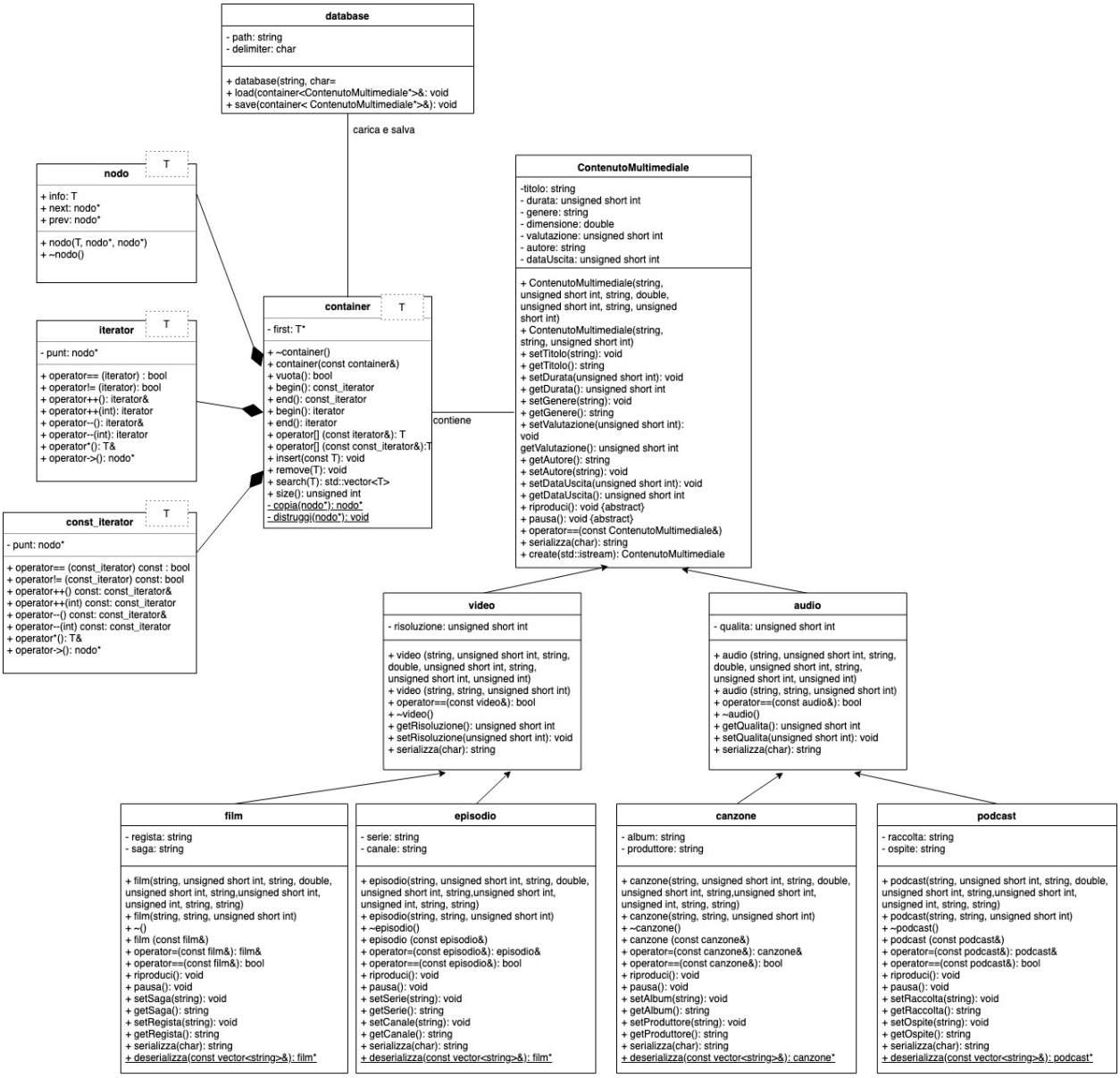

Figura 1: Diagramma delle classi del modello

- nelle finestre di aggiunta dei contenuti e di ricerca, all'interno della funzione `aggiungiInput()`, necessaria per caricare i box di testo e le rispettive etichette necessarie per il tipo di contenuto multimediale selezionato precedentemente nel menù a tendina.

- dentro al file `qontainer.cpp`, necessario per il corretto caricamento di tutte le finestre e dell'eliminazione della finestra corrente per far caricare correttamente la finestra successiva: nella funzione `eliminaContenutoAttuale()` viene usato un `dynamic_cast` necessario per la corretta eliminazione del *layout* attuale, effettuando un *downcast* a `QWidgetItem`.

- dentro al file `risultatoricerca.cpp` viene usato il *downcasting* per visualizzare a schermo il tipo dinamico dei puntatori ad oggetti contenuti nel contenitore che è stato ritornato come risultato dalla funzione `search(ContenutoMultimediale*)`.

- nell'aggiunta di contenuto nel contenitore tramite GUI, attraverso la funzione `aggiungiElemento()` all'interno del file `aggiungi.cpp`, l'aggiunta viene eseguita tramite un puntatore di tipo statico `Contenuto Multimediale*`, ma tipo dinamico differente a seconda del tipo di file che si vuole aggiungere nel contenitore.

5 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()`:

- `load()`, tramite un'operazione di deserializzazione implementata ad hoc in ogni classe derivata concreta della gerarchia, permette la lettura del contenuto in ogni file `.txt`. C'è un file `.txt` per ognuna delle classi `film`, `episodio`, `canzone` e `podcast`, con relativa funzione di deserializzazione, che trasforma ogni riga contenuta in quei file in oggetto della relativa classe. Database si occupa di caricare questi oggetti nella libreria, in modo che possano essere gestiti dal contenitore templetizzato.

- `save()` invece si occupa del salvataggio di eventuali modifiche apportate tramite GUI ai file `.txt` sovramenzionati, tramite le relative operazioni di serializzazione, contenute nelle relative classi.

6 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