Università degli Studi di Padova
Programmazione ad Oggetti

Corso di in Informatica

Relazione di progetto

Gestione di una biblioteca personale

16 settembre 2025

Candidato:
Simone Zecchinato, simone.zecchinato.1@studenti.unipd.it
Matricola 2113189

Docenti:
Prof. Francesco Ranzato
Prof. Marco Zanella

Anno Accademico 2024-2025

Indice

1 Introduzione

Il progetto si poneva come obiettivo la realizzazione di un'applicazione per la gestione di una **biblioteca personale**. La biblioteca che ho deciso di rappresentare è quella di un appassionato e collezionista di contenuti e supporti multimediali. Il programma permette di aggiungere, modificare, rimuovere, ricercare e visualizzare i propri elementi. I contenuti gestibili comprendono audio e video digitali e file audio e video, mentre i supporti includono CD e DVD, ciascuno con proprietà specifiche e una rappresentazione grafica dedicata. I principali punti di forza sono: l'implementazione di un sistema di ricerca e filtraggio che consente di combinare criteri multipli (titolo, autore, anno e tipologia) in modo flessibile, restituendo solo i risultati pertinenti e la distinzione netta fra il modello logico e la parte grafica. Questo è stato realizzato sfruttando polimorfismo e pattern di progettazione, così da rendere l'architettura estensibile e mantenibile. Ho scelto questo progetto perché unisce il mio interesse per il mondo multimediale con la possibilità di approfondire concetti di programmazione a oggetti, in particolare il polimorfismo, l'uso dei visitor e il pattern singleton per la gestione dello storage.

2 Modello Logico

Il modello logico del progetto è organizzato attorno a due gerarchie principali: quella dei **contenuti multimediali** e quella dei **supporti fisici**. Nel diagramma UML mostrato in Figura 1 si può osservare come entrambe le gerarchie derivino da classi astratte comuni, rispettivamente `contenutoMultimediale` e `supportoMultimediale`.

Queste classi astratte definiscono gli attributi e i comportamenti di base delle entità rappresentate. Nella classe **base astratta contenutoMultimediale**, in particolare, tra gli attributi principali troviamo:

- **titolo**: il titolo del contenuto;
- **casa produttrice**: la casa di produzione associata;
- **autori**: l'elenco degli autori coinvolti nella produzione;
- **anno di pubblicazione**: l'anno in cui il contenuto è stato pubblicato;
- **durata**: la durata totale del contenuto multimediale espressa in secondi;
- **copertina**: l'immagine di copertina associata;
- **scaffale**: l'indicazione dello scaffale o della collocazione.

Per ciascuno di questi attributi è previsto un metodo *getter* e un metodo *setter*, che permettono di accedere e modificare i valori in modo controllato, garantendo l'incapsulamento e la coerenza dei dati.

Questa struttura a gerarchie permette di gestire in maniera uniforme e modulare sia i contenuti multimediali sia i supporti fisici, facilitando l'estensione del sistema per nuovi tipi di contenuti o supporti senza modificare il modello di base.

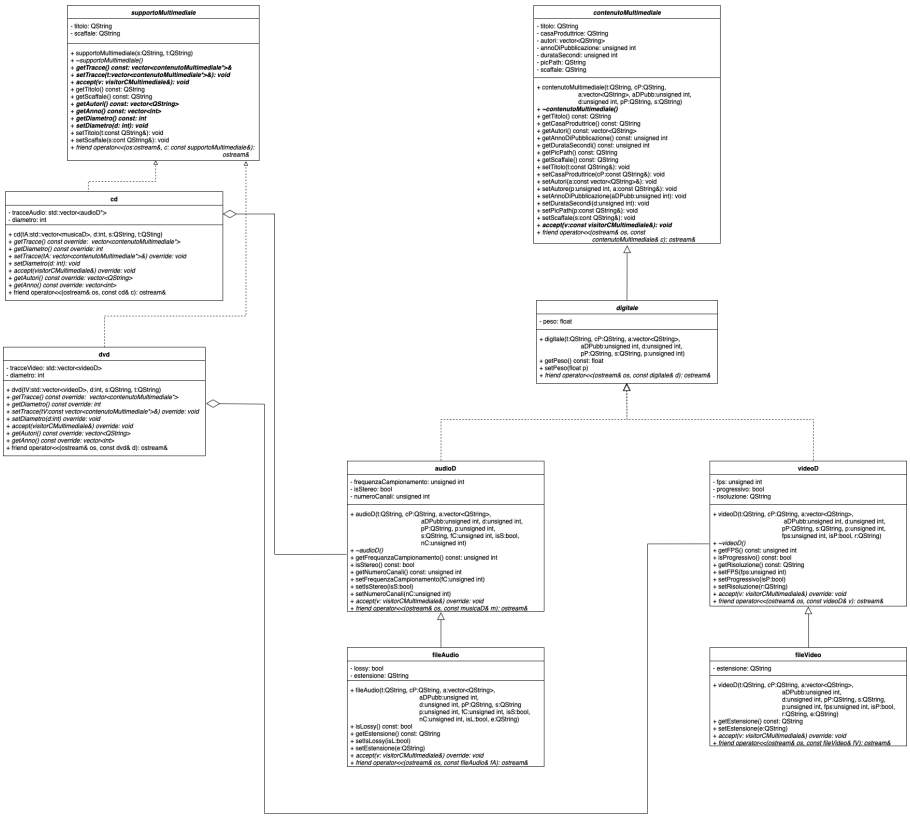

Figura 1: UML del modello logico

All'interno della gerarchia dei contenuti compare la prima sottoclasse astratta: la **classe digitale**. Essa introduce l'attributo fondamentale della natura digitale del contenuto e aggiunge poi l'attributo del peso. La struttura è stata progettata per essere estensibile: questa suddivisione, infatti, non vincola l'implementazione e permette, ad esempio, di aggiungere in futuro un ramo *analogico*, così da ampliare ulteriormente la gerarchia.

Troviamo poi, scendendo, le sottoclassi audioD, videoD, fileAudio e fileVideo. Questa scelta riflette la volontà di distinguere i contenuti digitali dai file multimediali locali, ognuno con proprietà specifiche (ad esempio, gli audio digitali hanno un numero di canali audio e i file audio aggiungono qualità ed estensione). In questo modo è possibile sfruttare il polimorfismo nelle operazioni di ricerca e visualizzazione, consentendo al programma di trattare uniformemente contenuti diversi ma con comportamenti specializzati. Per la **classe audioD** si aggiunge la frequenza di campionamento, il panorama stereo (o mono) e il numero di canali audio che contiene. Per la **classe videoD** si aggiungono: gli FPS, il tipo di scansione (se progressivo o interlacciato) e la risoluzione. **audioD e videoD** sono classi **concrete**, quindi implementano tutti i metodi virtuali puri della superclasse **contenutoMultimediale**. Scendendo nella **classe fileAudio** troviamo la compressione del file: lossy o lossless, e l'estensione; mentre per la **classe fileVideo** troviamo solamente l'estensione. Per tutti i campi sono previsti metodi *getter* e *setter*. Per scelta, anche se in modo un po' arbitrario, audio e video digitali possono comunque essere archiviati all'interno della biblioteca.

La seconda gerarchia riguarda invece i supporti, che comprendono cd e dvd. Questi oggetti non rappresentano singole tracce, ma sono container fisici capaci di includere più contenuti multimediali digitali, audio o video. La scelta di ciò che i supporti custodiscono riflette la realtà: un CD raccoglie un flusso di audio digitale, mentre un DVD ospita un flusso di video digitale. La distinzione tra contenuti e supporti deriva da una precisa decisione architetturale: in una biblioteca reale, infatti, è fondamentale separare ciò che viene fruito (audio o video) dal supporto che lo conserva (CD,

DVD). Sebbene i file audio e video possano essere visti come supporti, è più corretto considerarli figli diretti dei contenuti digitali, poiché un file multimediale contiene un solo contenuto e ne amplia le possibilità. Diversamente, questo non sarebbe realizzabile. La **classe** `supportoMultimediale` è una classe virtuale pura che definisce l'interfaccia comune a tutti i tipi di supporti multimediali. Essa fornisce un insieme di metodi virtuali puri che devono essere implementati dalle classi derivate, garantendo così un accesso uniforme ai dati contenuti nei supporti.

In particolare, i metodi principali sono:

- `getTracce()` e `setTracce()`: rispettivamente per ottenere e impostare l'elenco delle tracce presenti nel supporto.
- `getAutori()`: restituisce tutti gli autori associati alle tracce del supporto.
- `getAnno()`: restituisce tutti gli anni di pubblicazione delle tracce contenute nel supporto.
- `setDiametro()` e `getDiametro()`: consentono di impostare e ottenere il diametro del supporto fisico, se applicabile.

Questa interfaccia è fondamentale perché consente di trattare in maniera uniforme diversi tipi di supporti multimediali (come CD, DVD) senza conoscere i dettagli specifici della loro implementazione. Ogni classe derivata deve fornire una propria implementazione concreta dei metodi virtuali puri, rispettando così il contratto definito dall'interfaccia.

La **classe cd** contiene dunque il flusso audio, che non è altro che un vettore di audio digitali, e contiene poi il diametro del supporto. La classe cd è concreta quindi implementa tutti i metodi virtuali puri della superclasse **supportoMultimediale**. La **classe dvd** contiene il flusso video, un vettore di video digitali, e il diametro. Anche dvd è una classe concreta quindi implementa tutti i metodi virtuali puri della sua superclasse.

Entrambe le gerarchie sono integrate attraverso il pattern Visitor (Fig. 2), utilizzato per implementare operazioni specifiche senza compromettere l'estensibilità del modello. Questo consente, ad esempio, di definire comportamenti diversi nella generazione delle preview grafiche a seconda del tipo di contenuto o supporto considerato, mantenendo disaccoppiata la logica di presentazione dai dati sottostanti.

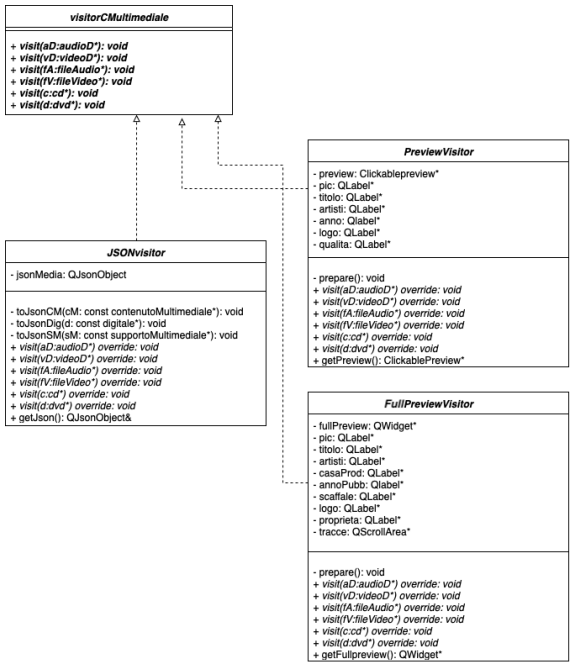

Figura 2: UML dei Visitor

Il modello è gestito dalla **classe biblioteca**, che fornisce tutti i metodi per gestire la biblioteca nella sua interezza. Questa classe comprende infatti i metodi fondamentali quali aggiunta e cancellazione. Essa è implementata tramite il **singleton pattern**: questa soluzione assicura un unico punto di accesso alla collezione di contenuti multimediali e supporti, evitando duplicazioni, garantendo coerenza nello stato interno e semplificando sia l'archiviazione che le operazioni di consultazione. La classe mantiene infatti i vettori di contenuti e supporti e delega alla **classe fileManager** le operazioni di persistenza su file JSON. Inoltre, consente la ricerca attraverso i metodi specifici realizzati utilizzando la **classe query**, con questi è possibile effettuare filtraggio e recupero mirato degli elementi.

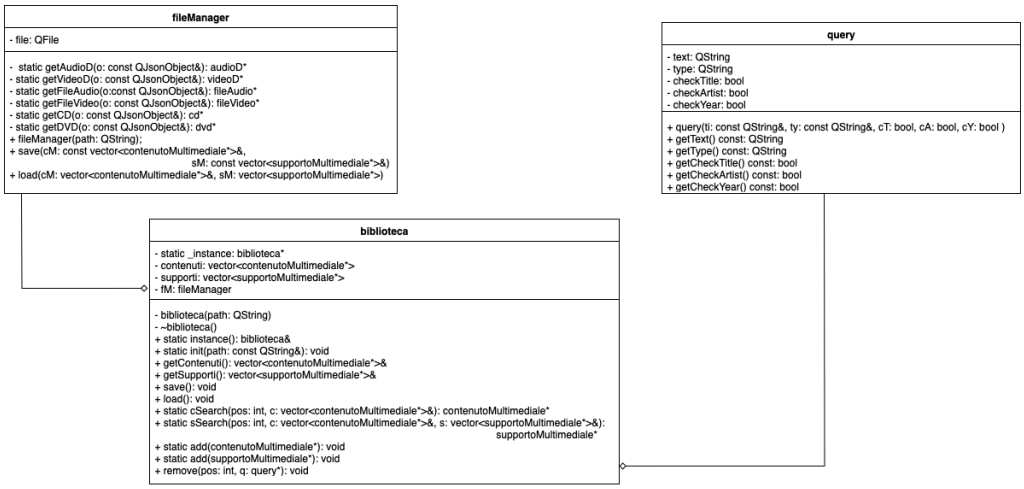

Figura 3: UML fileManager - biblioteca - query

3 Polimorfismo

L'utilizzo principale del polimorfismo nel progetto riguarda il **design pattern Visitor**, applicato alle gerarchie `contenutoMultimediale` e `supportoMultimediale` tramite l'interfaccia `visitorCMultimediale`. Questo approccio consente di definire operazioni specifiche sugli oggetti senza modificarne le classi concrete, garantendo un'estensibilità elevata e un forte disaccoppiamento tra la logica dei dati e quella della presentazione.

In particolare, il pattern Visitor è stato utilizzato per:

- **Serializzazione dei dati**: il visitor `JSONVisitor` gestisce la conversione dei contenuti e dei supporti in formato JSON, rendendo semplice l'integrazione con altre componenti o servizi esterni;
- **Creazione di preview grafiche**: il visitor `PreviewVisitor` costruisce anteprime compatte dei contenuti e dei supporti per l'interfaccia scrollabile, mentre il visitor `FullPreviewVisitor` genera una visualizzazione completa con tutti i dettagli;
- **Estensione delle funzionalità**: grazie alla struttura a visitor, è possibile aggiungere nuove operazioni sugli oggetti senza modificare le gerarchie esistenti, evitando controlli espliciti sul tipo dell'oggetto.

Ogni visitor definisce comportamenti specifici in base al tipo concreto dell'oggetto visitato, sfruttando pienamente il polimorfismo. Ad esempio, la generazione delle anteprime permette di avere diverse rese grafiche a seconda del contenuto o del supporto, senza introdurre condizionali complessi o duplicazioni di codice. Ciò garantisce coesione e manutenibilità: nuove tipologie di contenuti o supporti possono essere integrate semplicemente estendendo i visitor esistenti o

aggiungendone di nuovi. Questo utilizzo del polimorfismo si differenzia da quello più banale utilizzato nel modello logico per la definizione di gerarchie.

Nelle gerarchie troviamo il metodo virtuale puro `void accept(visitorCMultimediale&)` che è la porta d'accesso dei visitor agli oggetti. Dentro a questo metodo viene chiamato il metodo `visit(this)` che è comune a tutti i visitor che estendono *visitorCMultimediale*. Questo sistema permette di passare l'oggetto al visitor, che potrà usare l'interfaccia pubblica dell'oggetto(*getter o setter*) per visualizzarne o modificarne il contenuto.

In sintesi, l'uso del pattern Visitor rende il sistema altamente flessibile e pronto ad accogliere futuri sviluppi, valorizzando il polimorfismo come principio di progettazione e non come semplice strumento sintattico imposto dal linguaggio.

4 Persistenza dei dati

La persistenza dei dati nel progetto viene gestita attraverso il formato **JSON**, con un unico file per ciascun catalogo di contenuti e supporti. Il file di storage JSON contiene un vettore di oggetti che rappresentano i diversi contenuti multimediali e supporti fisici presenti nel catalogo. Gli oggetti sono principalmente associazioni chiave-valore, e per gestire correttamente le sottoclassi viene aggiunto un attributo `tipo` che ne identifica il tipo concreto.

La serializzazione in JSON è stata implementata tramite il **Visitor** `JSONVisitor`, che visita ciascun oggetto del modello e ne costruisce la rappresentazione JSON. In particolare:

- `toJsonCM` si occupa di serializzare gli attributi comuni a tutti i contenuti multimediali, come titolo, casa produttrice, autori, anno di pubblicazione, durata, percorso dell'immagine di copertina e scaffale;
- `toJsonDig` aggiunge le informazioni specifiche dei contenuti digitali, come il peso;
- `toJsonSM` gestisce le tracce dei supporti multimediali, iterando sui contenuti e richiamando i visitor appropriati in base al tipo di contenuto (audio o video);
- I metodi `visit` per ciascuna sottoclasse (`audioD`, `fileAudio`, `cd`, `videoD`, `fileVideo`, `dvd`) completano la serializzazione aggiungendo gli attributi specifici di ogni tipo, come frequenza di campionamento, numero di canali, estensione, diametro del supporto, FPS o risoluzione.

Il JSON così generato può essere salvato su file e successivamente ricaricato per ricostruire gli oggetti in memoria, garantendo persistenza dei dati e facilità di integrazione con altri sistemi.

A titolo di esempio, la Figura 4 mostra uno snapshot di un pezzo del file di storage fornito col progetto.

```
{
    "contenuti": [
        {
            "autore0": "Dire Straits",
            "canali": 2,
            "casaProduttrice": "Vertigo Records",
            "durata": 240,
            "frequenza": 44100,
            "peso": 100,
            "picPath": "src/copertine/sultans_of_swing.jpg",
            "pubblicazione": 1978,
            "scaffale": "A013",
            "stereo": "true",
            "tipo": "audioD",
            "titolo": "Sultans of Swing"
        }
    ],
    "supporti": [
        {
            "diametro": 12,
            "scaffale": "SA2",
            "tipo": "CD",
            "titolo": "Compilation",
            "tracce": [
                {
                    "autore0": "The Police",
                    "canali": 2,
                    "casaProduttrice": "MJD Records",
                    "durata": 240,
                    "frequenza": 44100,
                    "peso": 1000,
                    "picPath": "src/placeholder.png",
                    "pubblicazione": 1977,
                    "scaffale": "",
                    "stereo": "true",
                    "tipo": "audioD",
                    "titolo": "Message in a bottle"
                },
                {
                    "autore0": "Dire Straits",
                    "canali": 2,
                    "casaProduttrice": "Vertigo Records",
                    "durata": 240,
                    "frequenza": 44100,
                    "peso": 100,
                    "picPath": "src/placeholder.png",
                    "pubblicazione": 1978,
                    "scaffale": "",
                    "stereo": "true",
                    "tipo": "audioD",
                    "titolo": "Sultans of Swing"
                }
            ]
        }
    ]
}
```


Figura 4: Snapshot di un pezzo di file di storage

5 Funzionalità implementate

All'apertura il programma apre una finestra di dialogo e chiede si scegliere un file di storage. Può essere scelto qualunque file JSON vuoto o presalvato che sia. Nella cartella, a titolo di esempio è già presente il file `storage.json` con lo storage di esempio.

Scelto il file si apre l'interfaccia del programma vera e propria. Si possono notare quattro sezioni principali:

- **Sezione preview**: mostra la preview di tutti i contenuti e i supporti presenti nella biblioteca;
- **Sezione statistiche**: mostra le statistiche relative alla biblioteca (come numero di contenuti per tipo e numero totale);
- **Sezione cerca**: mostra l'interfaccia di ricerca fra i contenuti e i supporti e i vari filtri;
- **Sezione pulsanti**: mostra i pulsanti con le quattro funzioni principali del programma:
  - **Add**: mostra l'interfaccia per aggiungere un contenuto o un supporto;
  - **Remove**: attiva la modalità di rimozione, confermata da una piccola label sotto i pulsanti;
  - **Save**: salvataggio manuale della biblioteca;
  - **Reload**: ricarica dello storage all'ultimo salvataggio.

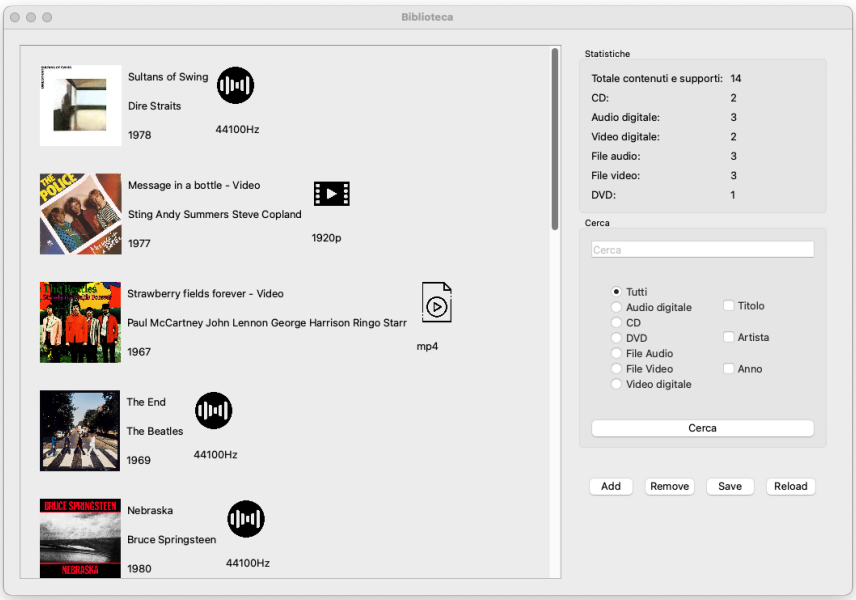

Figura 5: Schermata principale

Ogni *Preview* è cliccabile e se cliccata, in modalità normale, apre la *FullPreview*. Questa mostra tutte le informazioni del contenuto o del supporto. Se il programma è in modalità *"Rimozione"* (cliccando il tasto remove, visibile tramite una scritta sotto i pulsanti che avvisa l'utente) sarà possibile eliminare gli oggetti cliccando sulle *Preview*. Il programma lancerà una finestra di dialogo chiedendo conferma dell'eliminazione. La modalità rimane attiva per facilitare cancellazioni multiple. Dopo le modifiche è possibile effettuare un salvataggio manuale o un ricaricamento se si vuole fare un *"rollback"*. **Attenzione:** Alla chiusura il programma effettua di default il salvataggio delle informazioni.

Cliccando su *"Add"* si apre una schermata che chiede all'utente di scegliere cosa aggiungere:

- Contenuto: immagine a sinistra cliccabile;
- Supporto: immagine a destra cliccabile.

Aggiungendo un contenuto multimediale vengono innanzitutto richiesti i dati del contenuto, poi bisogna selezionare il tipo e completarne i campi specifici. Aggiungendo un supporto, invece, viene richiesto di aggiungere il titolo, lo scaffale e il diametro, poi di selezionare il tipo. A questo punto si potranno creare le tracce da masterizzare all'interno del supporto. Cliccando il tasto *"+"*, si verrà indirizzati all'aggiunta del contenuto multimediale, in cui però non sarà possibile selezionare il tipo. Questo è obbligatorio per evitare supporti con contenuti misti: CD con video o DVD con audio. Cliccando su *"Salva"* verremo riportati alla pagina principale e le aggiunte saranno effettuate.

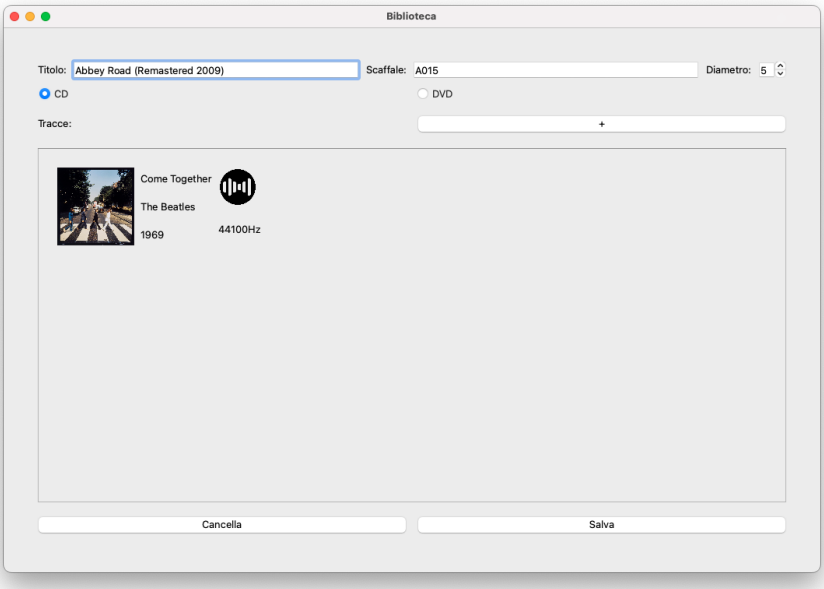

Figura 6: Esempio di aggiunta supportoMultimediale

6 Rendicontazione delle ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	12	12
Sviluppo del codice del modello	8	10
Studio del framework Qt	10	12
Sviluppo del codice della GUI	10	15
Test e debug	6	7
Stesura della relazione	4	4
totale	50	**60**

Il monte ore è stato superato del 20% in quanto lo sviluppo del codice del modello ha richiesto un paio di ore più del previsto. Lo studio del framework Qt ha sforato il previsto in quanto ho preferito fare delle prove prima di lavorare col codice del progetto. Lo sviluppo della GUI ha richiesto più ore del previsto per ulteriori prove e sistemazione degli errori e per prendere confidenza con il framework. La parte di test e debug ha richiesto un'ora più del previsto perchè ho dovuto adattare il codice e renderlo cross-platform, lavorando con macOs e rendendolo compatibile a pieno con Windows e Linux.