Relazione Progetto Biblioteca Virtuale

Brunello Marco
Matricola: 2110997

Anno Accademico 2024-2025

Contents

1 Introduzione

Il progetto verte nella creazione di una biblioteca virtuale.
Io ho deciso di implementare la mia versione ispirandomi ad un design che ho trovato nel web.

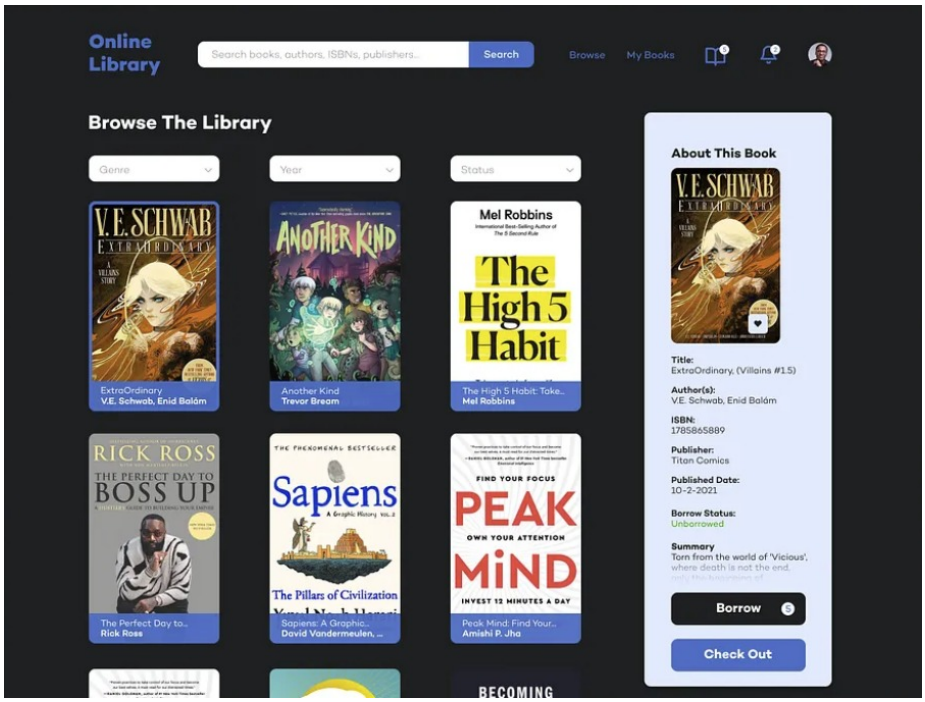

Figure 1: Design di riferimento

Per l'implementazione del progetto ho usato il framework Qt e alcuni design pattern come Visitor e Observer per rendere l'interfaccia grafica più dinamica e interattiva.
Il modo in cui ho gestito il progetto è stato quello di suddividere il progetto in interfaccia grafica e modello logico, dove il modello logico è responsabile di tutte le elaborazioni fondamentali del progetto e dà una struttura base su cui poi mi appoggio per la creazione dell'interfaccia grafica. Ciò che ritengo il punto forte del mio progetto è la dinamicità dell'interfaccia grafica e la potenza della feature di ricerca in quanto ho usato le Regular Expression.
Le Regular Expression sono delle stringhe dove specifico il pattern che voglio che altre stringhe, come per esempio quelle di input, rispettino.

2 Modello Logico

Lo scopo del modello logico è quello di essere del tutto indipendente dall'interfaccia grafica, per questo l'ho implementato prima della GUI.
Per la gestione degli elementi della biblioteca ho creato una classe base astratta "ElementoBiblioteca": questa contiene tutti gli attributi e metodi in comune dei diversi tipi di elementi.
Grazie alla classe "ElementoBiblioteca" ho creato 3 classi concrete che ereditano direttamente da lei. Queste classi sono "Libro", "Film" e "Brano" (Canzone).
La biblioteca in sé l'ho rappresentata tramite una classe che al suo interno contiene un "QVector<ElementoBiblioteca*>" che contiene tutti gli elementi caricati e gestiti in una particolare istanza della biblioteca. Vorrei però precisare che il QVector non è stata la mia scelta iniziale. In origine avevo optato per una QMap, con l'intenzione di sfruttare l'efficienza delle tabelle hash nelle operazioni di ricerca. Tuttavia, mi sono poi reso conto che, nel mio caso specifico, avrei comunque dovuto iterare l'intera struttura dati per trovare gli elementi. Per questo motivo ho preferito passare a un QVector: questa struttura memorizza gli elementi in celle contigue di memoria, il che può migliorare le performance durante le iterazioni grazie al caching dei dati, poiché più blocchi di memoria adiacenti possono essere caricati nella cache contemporaneamente.

3 Polimorfismo

Per rispettare il vincolo del polimorfismo non banale ho sfruttato i design pattern Visitor e Observer.

3.1 Visitor

Il design pattern visitor prevede la creazione di una classe base astratta, che nel mio caso è "ElementoBibliotecaVisitor", che fornisce un metodo "visit" per ogni classe concreta di interesse al visitor, sfruttando il best match del compilatore si riesce così a controllare il flusso di esecuzione facendo sì che venga eseguito il codice corretto in base al tipo dinamico del parametro passato; senza fare eccessivo uso dei dynamic_Cast.
Io ho impiegato il visitor principalmente nell'interfaccia grafica per far sì che venga mostrata la detail-view e edit-view corretta secondo il tipo dinamico/concreto dell'elemento.

3.2 Observer

Il design pattern observer prevede anch'esso la creazione di una classe base astratta solo che, invece del metodo "visit", abbiamo il metodo "notify".
Quest'ultimo ha il compito di "notificare" tutte le istanze di un observer che

"osservano" un determinato elemento della biblioteca per vedere se vi è stato un aggiornamento dello stato dell'oggetto e, in caso di aggiornamento, di comportarsi in un determinato modo.
Si intuisce dunque che questo design pattern è d'obbligo in ambito grafico, in quanto non appena un elemento esistente subisce una modifica, invece di richiamare un metodo di refresh della UI, l'observer aggiorna automaticamente e in real-time la UI. Ho infatti impiegato in questo modo l'observer.

3.3 Metodi "toJson()" e "fromJson()"

Spiegherò meglio nella sezione di persistenza dei dati, però questi metodi sfruttano il polimorfismo dunque ritengo opportuno almeno accennare il loro funzionamento.
Ho banalmente dichiarato due metodi virtuali "toJson()" e "fromJson()" nella classe base astratta "ElementoBiblioteca". Questi metodi, come si può intuire, hanno rispettivamente il compito di convertire e ritornare l'oggetto di invocazione convertito in un QJsonObject e riconvertire da QJsonObject a "ElementoBiblioteca".
Ogni classe concreta che eredita questi due metodi deve necessariamente effettuare l'override di questi due metodi aggiungendo i propri attributi alla conversione e riconversione.

3.4 Metodo "toString()"

Questo è un metodo virtuale della classe base astratta "ElementoBiblioteca", il quale scopo è quello di ritornare una stringa che descrive l'oggetto di invocazione tramite i parametri, utili per la ricerca, che lo compongono.
Sulla stringa ritornata da questo metodo verifico se vi è il match con la regular expression costruita con i parametri di ricerca inseriti dall'utente.

3.5 Valore Aggiunto

Queste funzionalità polimorfiche permettono, grazie all'override, di avere codice semplice da estendere, leggere e mantenere nel caso di un possibile ampliamento futuro dei media della biblioteca.

4 Persistenza dei dati

Per assicurare la persistenza dei dati ho optato il formato Json in quanto non sarebbe stata la prima volta che l'avrei impiegato.
Per evitare troppa complessità e eccessiva verbosità del codice: ho creato una classe chiamata "JSONController", il cui scopo è quello di salvare e caricare da file json la biblioteca, delegando così alle classi dei media della biblioteca il compito di convertire e riconvertire l'oggetto in Json, tramite i metodi virtuali

"toJson()" e "fromJson()".
"JSONController" è molto semplice nella sua struttura: non possiede attributi, bensì due metodi statici "loadFromFile(Bibiblioteca&, const QString&)" e "saveOnFile(Biblioteca&, const QString&)". Come si può capire dalla firma dei metodi: questi prendono un riferimento ad un oggetto di tipo "Biblioteca" e una "QString", che rappresenta il percorso del file.
Grazie alla delegazione, alla separazione e all'incapsulamento delle classi, non ho avuto nessun tipo di problema con Json quando è stata ora di eseguire le funzioni di salvataggio del modello logico su richiesta dell'utente tramite shortcut di salvataggio o caricamento da file.

5 Funzionalità Aggiuntive

5.1 Ricerca Dinamica

Come ho accennato in precedenza per la ricerca dinamica generale e specifica ho impiegato le regular expression.
Nel mio caso specifico ho creato la stringa di pattern partendo da input forniti dalla barra di ricerca e per ogni elemento della biblioteca mi facevo restituire una stringa che rappresentasse l'elemento con tutte le informazioni utili per la ricerca (metodo "toString()"), successivamente guardavo se la stringa dell'elemento rispettasse il pattern, ovvero avesse elementi che l'utente ha cercato.
Dopo una serie di ricerche sono riuscito a creare una regular expression in grado di adattarsi in base alla specificità della ricerca. Infatti la mia filosofia di ricerca è questa: di default cerca match dell'input su tutti i campi dell'oggetto, a meno di specifiche istruzioni. Questo perché ho cercato di rendere la ricerca la più realistica possibile in quanto, per esempio, se nella barra di ricerca di Netflix scrivessi il titolo di una serie tv o film, questo ovviamente me lo trova, ma se scrivo, per esempio, "Tom Cruise", senza dover cliccare checkbox o comunque qualcosa che notifica l'applicazione di Netflix che voglio cercare per attore, mi aspetto comunque che trovi tutti i film in cui Tom Cruise ha recitato.

5.1.1 Ricerca Generica

Come accennato la ricerca generica è una ricerca generale senza specificità di campo, in quanto non è realistico che un libro abbia la stessa stringa all'interno del campo di titolo e di autore, e anche se fosse, comunque la ricerca non sa per cosa effettivamente l'utente sta cercando (in quanto non è stato specificato) fornendo lo stesso l'elemento cercato.

La regular expression per la ricerca di tutti i campi è: `(?=.*<input>).+$`"
Un esempio:

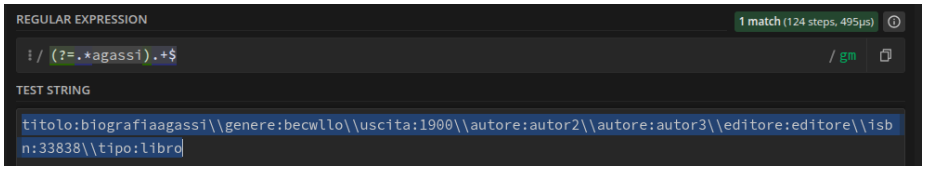

Figure 2: Ricerca su tutti i campi della presenza di "agassi"

La stringa più in basso è ciò che ritorna il metodo "toString()" nel caso l'elemento della biblioteca avesse "Libro" come tipo dinamico.
Le regular expressions sono come un linguaggio a parte e in quanto tale hanno una loro sintassi, in questo esempio specifico:

1. `(?=)`: Questo si tratta del "positive lookaround", ha il compito di assicurarsi che quanto segue sia presente;
2. `.*`: Questo significa qualsiasi carattere zero o più volte, escluso il carattere di nuova riga;
3. `agassi`: letteralmente stringa agassi;
4. `.+$`: Questo è il corpo principale della regular expression, dove `+` significa uno o più, mentre `$` indica la fine della stringa

5.1.2 Ricerca Specifica

Come accennato in precedenza, è possibile anche ricercare / filtrare per campi specifici.
Per poter filtrare non è necessario dover indovinare quale bottone o menù apre i filtri, è sufficiente scrivere sulla barra di ricerca il campo e la stringa di ricerca separati da ":" in questo modo:

In questo modo creo una regular expression che ricerca gli elementi che nella stringa descrittiva contengono "agassi" nella sezione del titolo.

La regular expression che viene generata dal programma è:

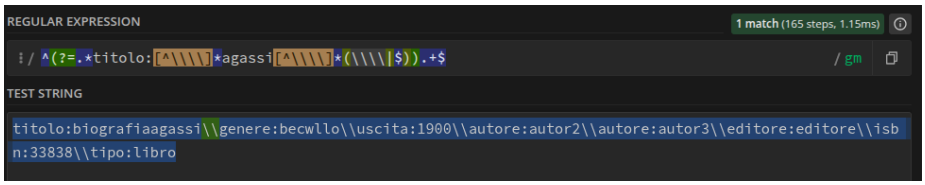

Figure 3: Esempio di regular expression di ricerca per titolo che contiene la parola "agassi"

Il significato di quest'ultima regular expression, a parte per i simboli che sono apparsi anche nella precedente regular expression, è:

1. `[^\\\\]*`: zero o più caratteri che non sono backslash;
2. `(\\\\|$)`: backslash o fine stringa;

Di fatto, grazie a questi ulteriori simboli, sto facendo come uno split della stringa dove incontro `\\`. Assicurandomi così che solo dopo a "titolo: agassi" vi siano i caratteri di escape `\\` e non, per esempio, nel mezzo tra i due.

5.1.3 Filtri Multipli

Se nella barra di ricerca, dopo aver selezionato l'input della ricerca, si inserisce il carattere virgola allora è possibile inserire un nuovo input di ricerca.
Non ci sono restrizioni per quanto riguarda la forma, ovvero si può combinare una ricerca generica con una ricerca specifica, sempre separati da virgola.

5.2 Scorciatoie Da Tastiera

Per migliorare l'esperienza utente ho implementato delle scorciatoie da tastiera.

1. Ctrl-s: Salva, questa scorciatoia salva la biblioteca senza dover aprire il QFileDialog;
2. Invio: Quando si scrive nella barra di ricerca si può premere invio per confermare l'input di ricerca;
3. Esc (Nella AddView): Quando si è nella "AddView" si può premere esc per annullare l'aggiunta e ritornare nella "MainView", cancellando tutti gli input precedenti.

4. Ctrl-o: Apre un QFileDialog dove si può scegliere il file .json da importare, una volta scelto la ui si aggiorna tramite un signal che indica alla "ProductsView" di cambiare gli elementi dai vecchi ai nuovi.

6 Tempo di implementazione previsto vs effettivo

Prima di iniziare a scrivere il codice del progetto ho proceduto ad una fase iniziale di "progettazione concettuale" delle classi e in generale di come volevo fare il progetto.
Successivamente ho implementato il modello logico e immediatamente dopo l'interfaccia grafica, infine ho fatto una sessione di refactoring per sistemare i warning durante la compilazione più migliorie di usabilità.
Per completare il progetto ho impiegato circa 82 ore totali, che sono suddivise come seguono.

6.1 Progettazione Concettuale

Per la progettazione concettuale ho fatto un grossolano schema delle classi in UML, giusto per rendermi conto dei metodi che avrei dovuto implementare e come strutturare la gerarchia delle classi.
Inizialmente pensavo di fare tutta la progettazione concettuale in circa 2 ore e mezza, invece ho contato che ci ho impiegato 3 ore e mezza, questo dovuto al fatto che ero piuttosto indeciso su come risolvere alcuni problemi di polimorfismo non banale.

6.2 Modello Logico

Per l'implementazione del modello logico ho stimato di impiegarci dalle 7 alle 10 ore.
Il tempo di implementazione effettivo del modello logico è di 15 ore, in quanto è proprio durante questa fase, circa alla ottava ora, che mi è venuta in mente l'idea di utilizzare le Regular Expression, che hanno portato ad un allungamento non del tutto insignificante, in quanto ho dovuto fare un po' di ricerche per comprendere i vari simboli di cui avrei avuto bisogno.

6.3 Interfaccia Grafica

L'interfaccia grafica è stata senza alcun dubbio la porzione di codice che ha preso più tempo di tutte, in quanto ambivo a riprodurre il design che ho fornito (vedi figura 1). Infatti ho fatto 53 ore, che ho suddiviso in:

- ToolBar: La ToolBar è stata la porzione dell'interfaccia grafica che ha portato via il meno tempo di tutte, infatti ho impiegato circa 3 ore ad implementarla;

- AddView: La Addview, dal punto di vista grafico, ho impiegato 5 ore a farla, considerando le shortcut che la compongono e i numerosi problemi di formattazione dell'immagine che ho riscontrato;

- ProductsView / ProductCard: Per l'implementazione di ProductsView e ProductCard insieme ci ho impiegato 7 ore, essendo ProductCard un observer ho dovuto infatti implementare un override del metodo notify, ovviamente, dato che sono presenti immagini, ho avuto difficoltà con la formattazione delle immagini e l'aggiustamento della ProductsView quando si ingrandisce o rimpicciolisce la finestra dell'applicazione;

- DetailView / EditView: Queste due "viste" sono direttamente collegate dato che sono entrabe dei visitor, dunque il loro contenuto varia a seconda del tipo dinamico dell'elemento selezionato.
  Ciò nonostante il fatto che sono delle viste a comparsa ho riscontrato una serie di problemi di memoria qualora avessi voluto farle comparire, cliccando sull'elemento.
  Ho stimato che la loro implementazione mi fosse costata dalle 10 alle 20 ore, data la complessità della logica del menù a comparsa che non si comportava come volevo io, ho impiegato 18 ore per implementare correttamente la parte grafica della DetailView e della EditView.

- MainView / MainWindow: Queste classi fungono da contenitore e da intermediario per i collegamenti di segnali e slot tra le diverse parti dell'interfaccia grafica.
  Inizialmente ho previsto, sottovalutando, che avrei impiegato al massimo 10 ore per entrambe, quando invece ho impiegato 17 ore e mezza ad implementare entrambe le classi, sempre dal punto di vista grafico, ingnorando i collegamenti

- Collegamento funzionalità logiche-grafiche: Questa parte è stata la parte che mi ha portato via più tempo di tutte, in quanto ho dovuto fare piccoli ritocchi anche nel modello logico, perché mi sono accordo mancavano importanti metodi get / set. Inoltre mi sono reso conto che mi serviva un comodo punto di accesso, da dovunque nel codice, all'istanza dell'oggetto di tipo "Biblioteca" che l'applicazione visualizzava e modificava; per questo ho usufruito del design pattern Service Locator (situato nei file AppContext e UIContext).
  La porzione di collegamento ha allungato i tempi di sviluppo dato che continuavo ad aggiungere segnali e slot che inizialmente non avevo previsto.
  Il tempo di sviluppo di quest'ultima porzione si aggira alle 24 ore, quando avevo previsto 10 ore al massimo.

6.4 Sessione Refactoring

Una volta ultimate tutte le funzionalità del progetto ho notato che durante la compilazione il compilatore dava dei warning riguardo la covarianza dell'overload virtuale dell'operatore di assegnazione di "ElementoBiblioteca" con tutte le sue classi figlie, e alcune ambiguità nell'ordine di inizializzazione nei costruttori.
Ci ho messo circa 1 ora e mezza a risolvere tutti i warning.
Successivamente mi sono reso conto che, seppur sottili, erano necessarie delle migliorie per quanto riguarda l'usabilità.
In particolare ho aggiunto a fianco della barra di ricerca un pulsante che apre un QDialog che spiega come usufruire della ricerca in maniera specifica.
Inoltre ho notato che non c'erano elementi nella UI che suggerissero all'utente di dover cliccare sopra ad un elemento della griglia elementi per vederne le caratteristiche.
Non c'erano nemmeno indizi per quanto riguarda la possibilità di inserire l'immagine tramite il click della copertina, sia nella "AddView" sia nella "EditView", dando per scontato che una persona intuitivamente avrebbe cliccato sopra di essi per importare l'immagine.
Ho risolto il problema in modo semplice ma efficace: ho impostato il cursore in modo che, passando sopra questi elementi, si trasformi in una manina (pointer), segnalando così all'utente che sono cliccabili.
Ho raggiunto questo obiettivo implementando la classe CursorEventFilter, progettata per modificare lo stato del cursore. Assegnando un'istanza di questa classe al widget desiderato tramite la funzione installEventFilter, ho ottenuto l'effetto di cambiare il cursore in un puntatore (pointer), segnalando così all'utente che l'elemento è cliccabile.
Questa miglioria di usabilità, tra ricerche nella documentazione e correzione di bug, mi è costata circa 5 ore.