Relazione biblioteca virtuale

Matteo Mazzaretto

Anno accademico 2024/2025

Indice

1 Introduzione

Il progetto riguarda la gestione di una biblioteca virtuale
Ho deciso di attenermi alle specifiche del progetto inserite sfruttando diverse funzionalità del polimorfismo e dei pattern come il visitor pattern e l'observer
All'inizio è presente una scelta, parte admin o parte utente
La parte utente non richiede niente, la parte admin richiede nome utente e password admin (non modificabili) Nel main viene creato il JsonManager, il cui scopo è estrarre la lista di dati dal file JSON selezionato. Questa lista viene poi elaborata e passata alle finestre AdminArea e UserArea
Inoltre queste due classi hanno un puntatore allo QStackedWidget creato nel main (così come la LibraryManager) per spostarsi da una finestra all'altra e sono derivate da JsonObserver per applicare l'observer pattern Da notare che la lista estrapolata verrà poi ordinata in ordine lessicografico in base al titolo

2 Classi principali modello logico

Ho realizzato una struttura gerarchica con 8 classi, 3 astratte e 5 concrete
La prima classe, la superclasse astratta, è la classe Biblioteca, dalla quale derivano le ulteriori due classi astratte: Cartaceo e Multimedia
Come i nomi suggeriscono le due specializzazioni si riferiscono al tipo dell'oggetto (nel senso stretto non nel senso della programmazione)
La classe "Cartaceo" si estende in ulteriori due rami:
1) Riviste
2) Libri
Da quest'ultima deriva l'ultima classe concreta per la parte cartacea, la classe "Manga"
Dalla parte dei multimedia invece si hanno le rimanenti 2 classi concrete:
1) Film
2) Cd

3 Polimorfismo

Ho aggiunto diverse funzionalità che sfruttano il polimorfismo

3.1 Visitor

Ho inserito il visitor pattern per la parte della GUI in modo da visualizzare in maniera diversa ogni oggetto selezionato in un preciso istante
Esso permette di visualizzare correttamente i vari elementi di ogni oggetto in base al suo tipo dinamico e ai suoi parametri

3.2 Observer

Ho pure implementato gli observer, i quali, dopo aver fatto le funzioni di salvataggio automatico in seguito ad una modifica o ad una eliminazione, noti-

ficano il cambiamento alle classi UserArea e AdminArea con elenco ordinato lessicograficamente in modo da mantenere la coerenza

3.3 Json

La classe JsonManager permette di salvare i cambiamenti od eliminare degli oggetti sfruttando le funzioni presenti in essa, le quali vengono chiamate tramite un gioco di segnali e slot i cui collegamenti sono definiti nel main
Inoltre la funzione save() si basa su 8 overloading della stessa funzione in base al tipo dinamico dell'oggetto, seguendo un approccio dal basso verso l'alto castando staticamente (perché si è sicuri di poterlo fare) per salvare il tutto nel file Json

3.4 LibraryManager

Le funzioni per modificare l'oggetto sono tutte dipendenti dalle funzioni save() che compiono diverse azioni in base al tipo dinamico dell'oggetto, infatti sono presenti 8 overloading (uno per classe) che permette di salvare solo i parametri di quella determinata classe
La funzione viene chiamata dal basso verso l'alto sfruttando i staticcast
Sempre in questa classe è presente una mappa che associa il typeid degli oggetti alla funzione per creare il Menù, qui purtroppo non ho potuto applicare l'idea dell'overloading della stessa funzione perché i parametri sono di default nullptr e moltissimi controlli si basano sul parametro passato che, se esiste, setta il testo dei vari elementi ai parametri in quel momento grazie ai getter
Lo stesso discorso vale per le funzioni create, dato che queste addirittura non hanno parametri (non si può passare un elemento ancora da creare)

3.5 Valore aggiunto

Tutte queste funzioni polimorfiche permettono di semplificare il progetto rendendolo più estensibile a future classi nel caso in cui se ne vogliano aggiungere, permettendoci di non preoccuparci del tipo dell'oggetto ma più sulla sostanza della funzione
Questo è rappresentato dal fatto che usiamo gli staticcast dal basso verso l'alto poiché si sa già che la funzione su cui si sta lavorando ha un parametro di tipo derivato rispetto alle funzione che si chiamano

4 Persistenza dei dati

In base al file JSON selezionato dall'utente, la classe JsonManager legge tutto il file ed estrae la lista che popolerà gli oggetti delle classi, tenendo separate le aree admin e aree utenti
Per farlo si sfrutta la funzione loadBibliotecaListFromJson, che legge tutti gli oggetti presenti nel file e in base al campo "Classe" costruisce l'oggetto corretto
Invece, le funzioni per salvare gli oggetti modificati rileggono tutto il file fino a quando non trovano l'oggetto modificato giusto, modificano il suo QJSonObject e poi riscrivono tutto nel file .json in maniera corretta (purtroppo informandomi non c'era la possibilità di modificare solo quell'elemento come in un database

ma bisogna riscrivere tutto il file)
La funzione per eliminare un oggetto ha più o meno lo stesso funzionamento
Da notare che, tutte queste funzioni, modificano l'oggetto giusto anche se sono presenti più oggetti con lo stesso titolo sfruttando il controllo che, fino a quando non si trova l'oggetto giusto, se si trova uno con lo stesso titolo esso verrà skippato nel ciclo che modifica o elimina gli oggetti, in modo da impedire che se si volessero modificare due oggetti che condividono lo stesso titolo non modifichi sempre e solo il primo ma modifichi correttamente quello selezionato
Nel file Json già presente nella cartella sono già presenti delle copie di oggetti in modo da testare il corretto funzionamento La cosa comoda è che il salvataggio è automatico, e né l'utente né l'amministratore devono fare qualcosa ma è tutto un processo automatico grazie ai segnali

5 Funzionalità aggiuntive

5.1 Area utente e area admin

Ho deciso di tenerle come "sezioni completamente a parte" in modo da separare ciò che è presente e ciò che si può modificare, infatti in una normale applicazione i privilegi da utente e amministratore sono completamente diversi Ispirandomi ad applicazione o siti web realistici, ho deciso di dividere le funzionalità in due parti:

1. Area utente: permette di prenotare, restituire, guardare, leggere, ascoltare un oggetto, visualizzarli per tipo oppure cercare l'oggetto che si vuole magari prenotare. Inoltre è presente la funzionalità Simili che mostra gli oggetti che verosimilmente possono piacere ad una persona a cui piace quel determinato oggetto. Questa ricerca si basa su un algoritmo che suggerisce gli oggetti con stesso regista, autore, artista, genere o attore ma non guarda gli stessi titoli
   Infatti la Divina Commedia di Dante Alighieri e la Divina commedia di Tedua non sono suggerite in quanto non condividono delle vere e proprie similitudini (seppur La Divina commedia di Tedua si ispiri all'omonima opera di Dante)

2. Area admin: permette di eliminare, modificare e creare un nuovo oggetto, il tutto portando le modifiche nel file Json. Il menù cambia in base al tipo dell'oggetto che si vuole modificare/creare. Infatti, se si vuole modificare un oggetto sono presenti i campi d'inserimento con già i valori di esso, mentre se si vuole creare si crea un menù dinamico in base al tipo di oggetto selezionato dalla tendina posizionata in alto che si vuole creare. Le immagini inserite vengono copiate(se non presenti) nella cartella "IMG/" scrivendo il nuovo percorso relativo nel file Json, altrimenti se già presente sovrascrive col percorso relativo senza però ricopiare l'immagine
   Inoltre è da notare la gestione del "triangolo" fra disponibilità, copie e numero prestiti, infatti:

   (a) nella creazione dell'oggetto, se si seleziona "Non disponibile" automaticamente le copie disponibili agli utenti non possono essere ¿0 mentre il numero dei prestiti deve essere obbligatoriamente ¿=1 (una

biblioteca non può esporre qualcosa senza permettere agli utenti di poterlo prestare, infatti anche per questo di default il suo valore minimo è 1)

(b) se si seleziona "Disponibile", il numero minimo dei prestiti diventa almeno il numero delle copie che si vogliono scegliere, infatti una biblioteca non può permettere di prenotare più copie rispetto a quante effettivamente ne possiede

(c) nella modifica dell'oggetto, si possono modificare solamente il numero dei prestiti e non può andare sotto le copie disponibili (per lo stesso motivo sopra)

Hanno inoltre dei tasti nella QToolBar che:

1. Permette di tornare indietro alla MainWindow (la modifica/creazione di un oggetto chiamata dalla finestra dedicata a quello permette di tornare di nuovo all'adminArea per impedire il continuo inserimento di username e password)
2. Chiudere il programma, tasto molto banale ma utile nel contesto se si è a schermo intero

Queste due classi si basano principalmente sulle funzioni showAll, showTipi, cercaDigitato e anche suggerisciSimili per l'area utente
Sono gestite dai vari booleani/testi, infatti essi sono necessari per visualizzare la corretta schermata dopo un'operazione evitando di mostrare nuovamente la schermata completa
Se per caso tutti i booleani dei tipi sono **false** si avvia in automatico la funzione showAll(), questo controllo serve ad impedire la visualizzazione di una finestra con niente
Infine, quando si torna indietro alla schermata principale tutti i booleani sono reimpostati su false, il testo su stringa vuota e ciò permette che, quando si rientra nella finestra, mostra tutto invece di ciò che mostrava prima (come fosse un nuovo accesso)

5.2 Prenotazione, restituzione, disponibilità

Nella superclasse Biblioteca sono dichiarate ed implementate le funzioni

1. bool Prenota()
2. bool Restituisci()
3. bool getDisponibilita() const

Queste lavorano su un classico di una biblioteca: prendere in prestito i libri
Infatti ogni oggetto di tipo Biblioteca (e di conseguenza i suoi sottotipi) ha un bool disponibile (che è sempre true come da collegamento col "triangolo" descritto sopra), ha un determinato numero di copiedisponibili e un determinato numero di prestiti effettuabili

Il primo rappresenta le copie presenti in quel momento nella biblioteca, il secondo le copie presenti in quel momento + quelle in prestito
La funzione Prenota(),se il booleano disponibile è true, diminuisce il numero delle copie di 1 (non ho impostato la possibilità di prenotare più copie dello stesso oggetto contemporaneamente, seppur facile ma insensato per la parte di un utente) e aggiorna la disponibilità con una funzione dichiarata privata nella classe Biblioteca
Se la prenotazione è avvenuta con successo restituisce un true, altrimenti un false
Questo booleano permette alla GUI di gestire il corretto pop-up (la prenotazione ha avuto effetto o no)
La funzione Restituisci() è praticamente simmetrica, mentre la getDisponibilità() già fa capire dal suo nome cosa permette di fare
Ogni volta che si schiaccia su un pulsante per prenotare e restituire, se la funzione fa ill suo dovere (ovvero può cambiare effettivamente il campo dato) emette un messaggio di conferma e scrive tutto nel file Json sfruttando le funzioni savePrenota, saveLetto, saveAscoltato, saveCd con un gioco di collegamenti definiti nel main, altrimenti emette un messaggio di warning
Da notare che in base al tipo si avrà un pulsante diverso che cambia anche in base ai booleani e cambia anche il significato dei QMessageBox

5.3 Visto/Ascoltato/Letto

Le classi "Cd", "Film" e tutte le derivate di "Cartaceo" permettono all'utente di visionare se un oggetto è già stato letto/ascoltato/visto e permettono di modificare questa cosa tramite l'area utente (ma non la parte admin)
Funzionalità molto banale ma che integra molto bene le funzionalità di queste classi e gli scopi di una biblioteca (magari una persona non vuole riprendere in prestito un oggetto che ha già visto)

5.4 Ricerca dinamica

Sia l'area utente che l'area admin permettono una ricerca dinamica, infatti ad ogni tasto premuto nella barra di ricerca si visualizzano le cose che presumibilmente si stavano cercando con un algoritmo che le cerca per titolo, genere, regista, attore, autore o artista
Oltretutto è caseInsensitive e non serve scrivere il nome esatto grazie alla funzione QString::startsWith già integrata in Qt

6 Rendicontazione ore previste e svolte

Prima di mettermi completamente nel progetto ho guardato abbastanza a fondo gli esercizi cercando di aggiungere funzionalità non richieste in modo da comprendere al meglio come poter strutturare il mio progetto e le mie idee
Purtroppo, ci ho messo veramente molto di più del previsto, superando addirittura le 80 ore (secondo i miei calcoli sono arrivato ad 85)
Infatti ho pagato molto la mia inesperienza nel creare progetti, la mia smaniata cura dei dettagli (appena trovavo un minuscolo problema dovevo risolvere), la mia non comprensione piena dei vincoli da rispettare che mi hanno costretto

a cambiare circa un quarto del progetto e la mia grande ambizione delle mie funzionalità aggiuntive e delle 8 classi, che alla fine guardando il progetto ha decisamente giovato

6.1 Modellazione progetto

Avevo previsto circa due ore per il "brain-storming", la modellazione del diagramma UML del modello logico (presente in fondo alla relazione) e le idee delle varie funzionalità, in effettiva ci ho messo circa 4 ore all'inizio salvo poi tutti i cambiamenti nel corso del progetto che avranno occupato sì e no 3 ore (la decisione di come implementare le classi e cosa fargli fare, la decisione dei parametri const/non const, infatti la mia idea originale prevedeva dei parametri immutabili, ecc...)

6.2 Implementazione modello logico progetto+visitor pattern

Ho previsto per l'implementazione del modello logico all'incirca 10 ore, in quanto una volta ideato a fondo il diagramma si è più fluidi nella scrittura del codice delle classi e del polimorfismo, infatti con esso ho rispettato la durata, decidendo di usare le std::string invece che le QString in modo da "prevedere" un eventuale utilizzo su altre piattaforme

6.3 Interfaccia grafica

Decisamente la parte che mi ha occupato più tempo di tutte, all'incirca 40 ore, infatti la costruzione dei vari visitor, delle varie ereditarietà (QMainWindow, QWidget), della disposizione degli oggetti nella QGridLayout e della successiva creazione della nuova classe LibraryManager a causa della mancata comprensione dei vincoli sono stati i motivi principali per cui ci ho messo così tanto, per l'appunto ho dovuto all'incirca cambiare un quarto di progetto che pensavo di aver finito
Inoltre a metà dell'opera ho dovuto implementare l'Observer pattern per permettere di far comunicare la classe JsonManager con le due finestre admin e user in modo da "notificare" i cambiamenti del file e visualizzarli correttamente
Poi ha portato via tanto tempo tutta la gestione delle immagini, dei sfondi, dello stile, la gestione delle funzioni, la creazione delle struct nelle classi, la ricerca di come eseguire una determinata operazione (ad esempio non sapevo sfruttare i vari widget contenitori, i qobjectcast di cui ho sfruttato tantissimo l'utilità con una QMap nella LibraryManager)

6.4 JSon

Avevo previsto circa 7 ore a riguardo però nella realtà ci ho messo di più, all'incirca il triplo (una ventina di ore)
Le principali difficoltà riscontrate sono state la riscrittura del file, l'implementazione dell'ObserverPattern menzionato prima, le funzioni di modifica, creazione e salvataggio e le funzioni di load con l'implementazione di struct specifiche

6.5 Gestione segnali e slot

Anche questo mi ha portato via un po' di tempo, infatti dovevo capire quali classi dovevano comunicare fra di loro, come farle comunicare, i parametri di stesso tipo e la connessione corretta, tutte le connessioni fra classi diverse però sono inserite nel main

6.6 Correzione bug

Praticamente tutto il tempo rimanente, ovvero circa 10 ore (escludendo già ciò che consideravo come parte di Json e delle varie interfaccie) dei quali menziono alcuni:

1. La necessità di ordinare la lista presente anche nella classe JsonManager prima di avviare la funzione "notifyObservers" in modo che gli oggetti di tipo UserArea e AdminArea ricevano una lista già ordinata

2. I vari segmentation fault incontrati a causa dei qobjectcast errati, dei staticcast errati e dei mancati controlli sul puntatore valido o meno

3. I vari segnali che non funzionavano a dovere, oppure funzionavano troppo bene
   Infatti ho dovuto bloccare temporaneamente (come nella funzione showAll di AdminArea e UserArea) oppure forzare l'emissione (come nella creazione di un oggetto nella LibraryManager)

4. Le funzioni di clearLayout, effettivamente necessarie perché altrimenti si vedevano troppe cose sballate nello schermo e non ci si capiva niente di ciò che si stava vedendo

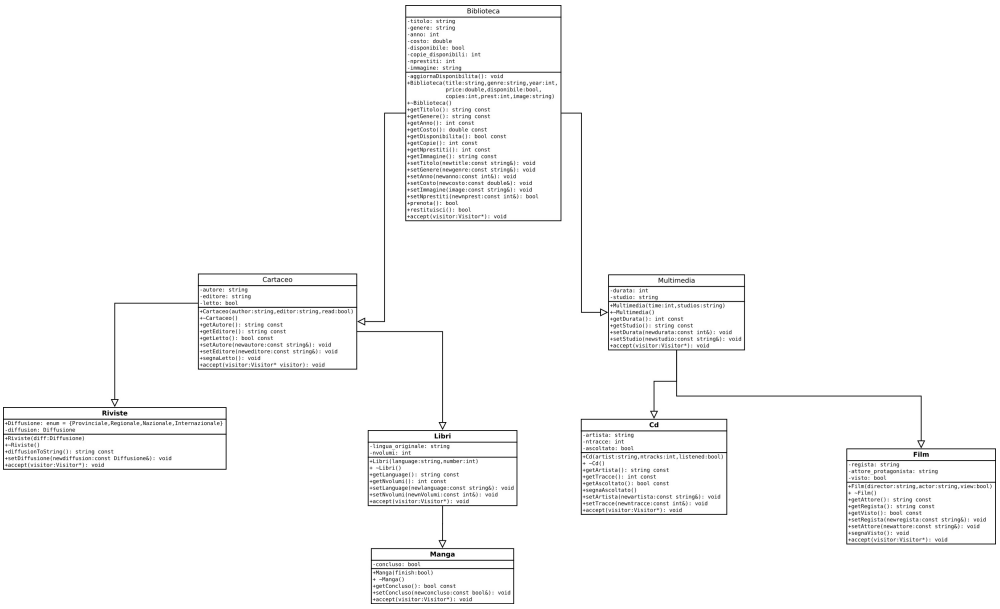
7 Modifiche

1. Sistemato un piccolo bug per cui non modificava correttamente lo stato di conclusione della classe concreta Manga

2. Sistemato un bug che causava un segmentation fault dove, a volte, compariva il pulsante salva quando si fanno in successione le modifiche e le creazioni di nuovi oggetto

3. Adesso la classe Visitor sfrutta appieno il polimorfismo, infatti, nell'eventualità dell'aggiunta di un visitor nel futuro, adesso le classi concrete richiedono un Visitor* che chiama la funzione accept in base al tipo dinamico del visitor, ciò ha portato alla conseguente creazione di variabili Visitor* nello heap e non più nello stack come in precedenza nelle funzioni di tipo show della GUI

4. Aggiunta la richiesta di conferma di eliminazione


Biblioteca
-titolo: string
-genere: string
-anno: int
-costo: double
-disponibile: bool
-copie_disponibili: int
-nprestiti: int
-immagine: string
+aggiornaDisponibilita(): void
+Biblioteca(title:string,genre:string,year:int,
price:double,disponibile:bool,
copie:int,prest:int,image:string)
+~Biblioteca()
+getTitolo(): string const
+getGenere(): string const
+getAnno(): int const
+getCosto(): double const
+getDisponibilita(): bool const
+getCopie(): int const
+getNprestiti(): int const
+getImmagine(): string const
+setTitolo(newtitle:const string&): void
+setGenere(newgenre:const string&): void
+setAnno(newanno:const int&): void
+setCosto(newcosto:const double&): void
+setImmagine(image:const string&): void
+setNprestiti(newnprest:const int&): bool
+prenota(): bool
+restituisci(): bool
+accept(visitor:Visitor*): void
Cartaceo
-autore: string
-editore: string
-letto: bool
+Cartaceo(author:string,editor:string,read:bool)
+~Cartaceo()
+getAutore(): string const
+getEditore(): string const
+getLetto(): bool const
+setAutore(newautore:const string&): void
+setEditore(neweditore:const string&): void
+segnaLetto(): void
+accept(visitor:Visitor* visitor): void
Multimedia
-durata: int
-studio: string
+Multimedia(time:int,studiss:string)
+~Multimedia()
+getDurata(): int const
+getStudio(): string const
+setDurata(newdurata:const int&): void
+setStudio(newstudio:const string&): void
+accept(visitor:Visitor*): void
Riviste
+Diffusione: enum = {Provinciale,Regionale,Nazionale,Internazionale}
-diffusion: Diffusione
+Riviste(diff:Diffusione)
+~Riviste()
+diffusioneToString(): string const
+setDiffusione(newdiffusion:const Diffusione&): void
+accept(visitor:Visitor*): void
Libri
-lingua_originale: string
-nvolumi: int
+Libri(language:string,number:int)
+ ~Libri()
+getLanguage(): string const
+getNvolumi(): int const
+setLanguage(newlanguage:const string&): void
+setNvolumi(newNVolumi:const int&): void
+accept(visitor:Visitor*): void
Cd
-artista: string
-ntracce: int
-ascoltato: bool
+Cd(artist:string,ntracks:int,listened:bool)
+ ~Cd()
+getArtista(): string const
+getTracce(): int const
+getAscoltato(): bool const
+segnaAscoltato()
+setArtista(newartista:const string&): void
+setTracce(newntracce:const int&): void
+accept(visitor:Visitor*): void
Film
-regista: string
-attore_protagonista: string
-visto: bool
+Film(director:string,actor:string,view:bool)
+ ~Film()
+getAttore(): string const
+getRegista(): string const
+getVisto(): bool const
+setRegista(newregista:const string&): void
+setAttore(newattore:const string&): void
+segnaVisto(): void
+accept(visitor:Visitor*): void
Manga
-concluso: bool
+Manga(finish:bool)
+ ~Manga()
+getConcluso(): bool const
+setConcluso(newconcluso:const bool&): void
+accept(visitor:Visitor*): void