

Relazione per progetto
“Programmazione ad oggetti”
Gestionale cinema

Arianna Pagano
Lorenzo Sansone
Alessandro Zirondelli

25 giugno 2021

Indice

1 Analisi	3
1.1 Requisiti	3
1.2 Analisi e modello del dominio.....	4
2 Design.....	6
2.1 Architettura.....	6
2.2 Design dettagliato.....	8
3 Sviluppo.....	34
3.1 Testing automatizzato.....	34
3.2 Metodologia di lavoro.....	35
3.3 Note di sviluppo.....	37
4 Commenti finali.....	38
4.1 Autovalutazione.....	38
4.2 Commenti per i docenti.....	39
A Guida utente.....	40

Capitolo 1

Analisi

Il nostro Team si è posto come obbiettivo la realizzazione di un applicativo per la gestione di un generico cinema. Il sistema permetterà di effettuare tutte le operazioni necessarie per monitorare ed organizzare il corretto svolgimento dell'attività. In particolare, l'utilizzo di questo software è stato pensato per due diversi contesti di impiego, amministrazione e personale, ognuno legato a proprie specifiche e permessi.

1.1 Requisiti

Requisiti funzionali

- L'applicazione all'avvio dovrà gestire l'autenticazione dell'utente, in base alle varie tipologie: amministratore e operatore.
- Il sistema permetterà di accedere ed effettuare operazioni tramite un apposito menu principale. In particolare l'amministratore si occuperà della gestione dei film e relative programmazioni, gestione delle prenotazioni e statistiche generali di vendita. Infine gestione account.
- L'operatore potrà accedere alla sola gestione delle prenotazioni.
- Si potranno gestire gli account registrati di entrambi i tipi, attraverso l'eliminazione e registrazione di nuovi utenti.
- Sarà resa possibile la registrazione di nuovi film all'interno del cinema inserendo le relative specifiche. Come titolo, durata, genere, descrizione e opzionalmente anche un'immagine di copertina. Dovrà essere data la possibilità di eliminare o modificare film esistenti.
- Si darà la possibilità di schedulare un film già inserito specificando i dati relativi alla programmazione, quali data, ora e sala di proiezione.
- Quando sarà eliminato un film dovranno essere rimosse tutte le relative programmazioni e i biglietti venduti.
- Quando sarà eliminata una singola programmazione dovranno essere rimossi i soli biglietti venduti per quello specifico film programmato.
- Si potranno visionare delle statistiche base relative all'intero cinema, come ad esempio il giorno e l'ora più affluente, il guadagno totale e il film più visto.
- Sarà possibile poter prenotare i posti disponibili per un dato film programmato in precedenza.
- Il costo di vendita del ticket può variare in diverse programmazioni, anche per lo stesso film.
- L'applicazione dovrà gestire il salvataggio dei dati in locale.

Requisiti non funzionali

- Il gestionale dovrà essere portabile su più piattaforme come Windows, Linux...
- L'interfaccia grafica deve essere il più intuitiva possibile per l'utente.

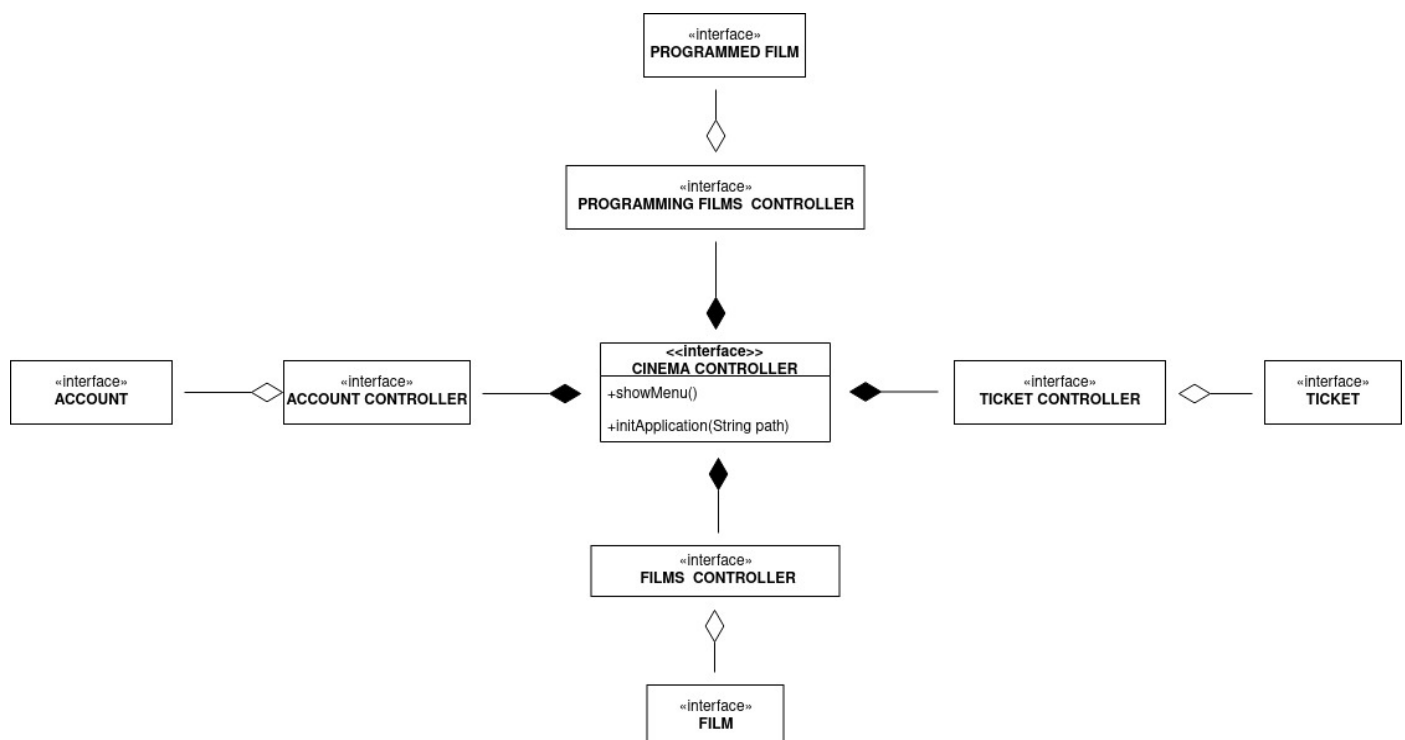
1.2 Analisi e modello del dominio

Analizzando il sistema che si punta a realizzare, si evince che il dominio applicativo è riconducibile alle seguenti entità base quali film, film programmati, account e vendita dei biglietti. Essi permettono la memorizzazione di tutte le informazioni principali di cui il nostro sistema necessita.

Il sistema dovrà essere in grado di permettere l'autenticazione di diverse tipologie di utenti. Le funzionalità che metterà a disposizione, dipenderanno dalla tipologia dell'account che si sarà autenticato, per cui sarà necessario implementare un sistema di caching dell'account loggato.

Per il corretto funzionamento dell'applicativo sarà di fondamentale importanza implementare delle politiche di gestione per ogni entità e funzionalità del nostro dominio per essere correttamente inserite all'interno del sistema.

Inoltre sarà necessario adottare un sistema di salvataggio dei dati per dare continuità di lavoro nel tempo.



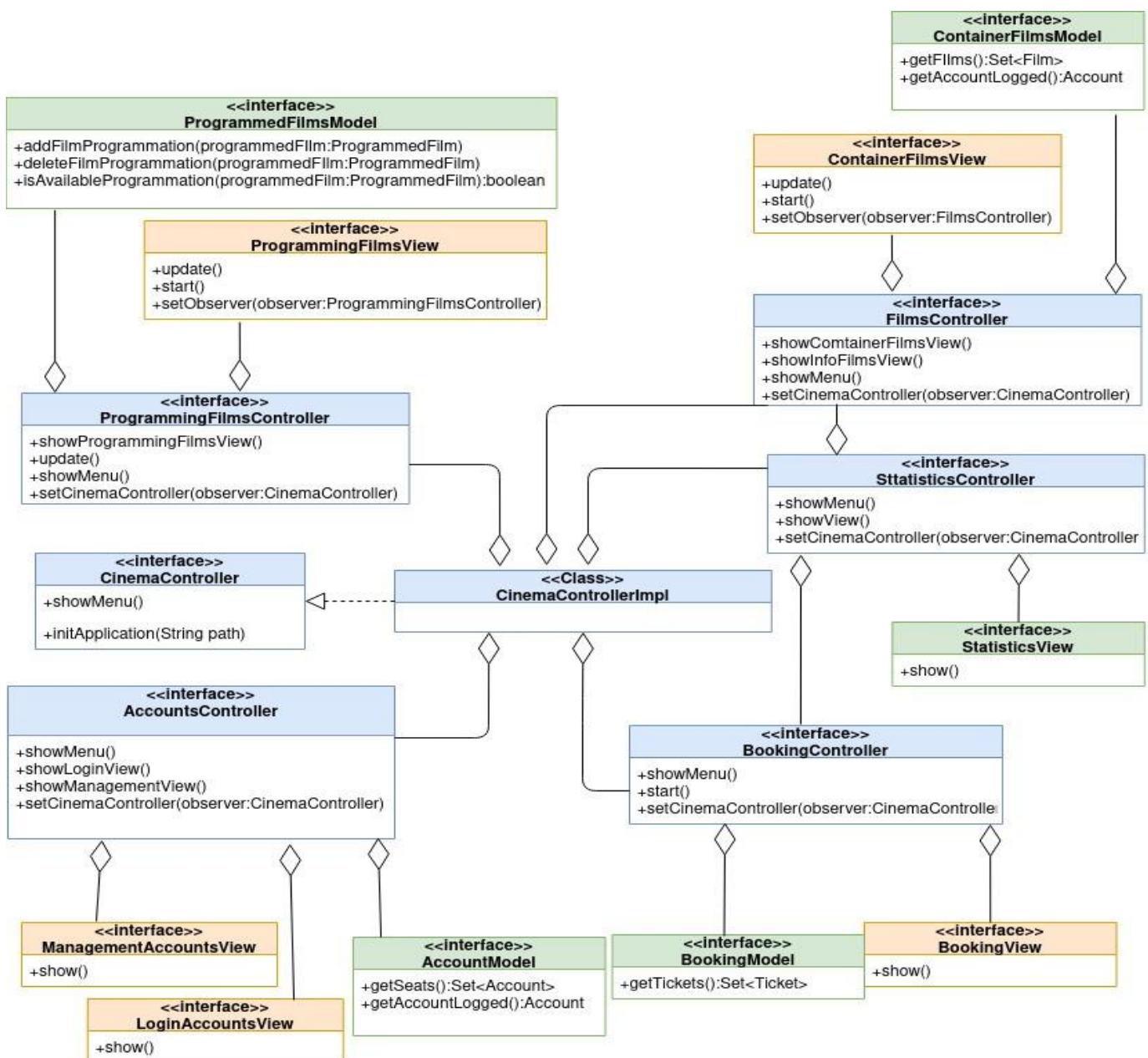
Per quanto riguarda la gestione degli account si denota la necessità di implementare dei controlli di unicità degli stessi. Invece per la gestione dei film risulta necessario concretizzare un meccanismo di importazioni delle immagini. Durante la fase di programmazione sono di fondamentale importanza i controlli relativi alla verifica della disponibilità della sala in un determinato giorno per una specifica fascia oraria. Infine sarà utile disabilitare la possibilità di prenotare i posti di programmazioni passate e di non permettere la prenotazione di posti già occupati.

Capitolo 2

Design

2.1 Architettura

Per lo sviluppo dell'architettura dell'applicativo, si è deciso di adottare il pattern architetturale MVC (Model-View-Controller). Si è scelto di utilizzare come “entry-point” un unico controller chiamato CinemaController, il quale attraverso controller più specializzati andrà a gestire l'intero funzionamento.



2.1.1 Model

Il Model si occupa della memorizzazione e dell'accesso ai dati del dominio applicativo. Rappresenta quindi lo stato attuale dell'applicazione e ne è il completo responsabile.

Data l'esistenza di più entità base all'interno del nostro dominio applicativo, è stata scelta la suddivisione in più model, ognuno relativo a una specifica entità e/o funzionalità.

2.1.2 View

La View è un insieme di componenti che costituiscono l'interfaccia grafica con la quale l'utente interagirà. Il suo compito principale è quello di visualizzare graficamente lo stato attuale del Model, notificando il controller in caso di cambiamenti apportati dall'utente. Tutta la componente View è stata pensata per essere completamente intercambiabile.

Questo significa che se in futuro si decidesse di implementare una nuova interfaccia grafica, lo si potrebbe fare evitando di causare cambiamenti sulle altre due componenti ossia Model e Controller.

Dato l'elevato numero di funzionalità disponibili, è stato necessario implementare differenti schermate in modo che ogni view gestisca una parte ben definita di operazioni e sia gestita da un determinato controller.

2.1.3 Controller

Il Controller è il componente che si occupa di fare da intermediario tra View e Model. Questo accade quando l'utente interagisce con la view e di conseguenza è necessario comunicarlo al Model attraverso il Controller e viceversa.

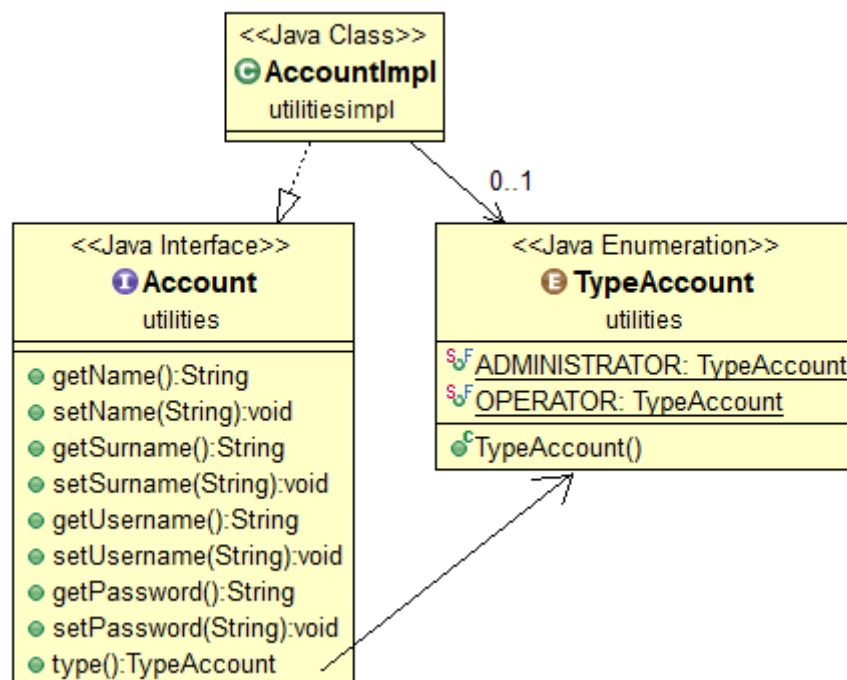
É stato scelto di utilizzare un controller principale (CinemaController) che gestirà ulteriori sotto-controller i quali avranno la responsabilità di specifiche funzionalità .

2.2 Design dettagliato

2.2.1 Arianna Pagano

La parte di cui mi sono occupata singolarmente è stata quella inerente all'accesso tramite login di Username e Password, l'intera gestione degli utenti con le relative operazioni e infine quella inerente alle statistiche base generali riferite al Cinema.

Account

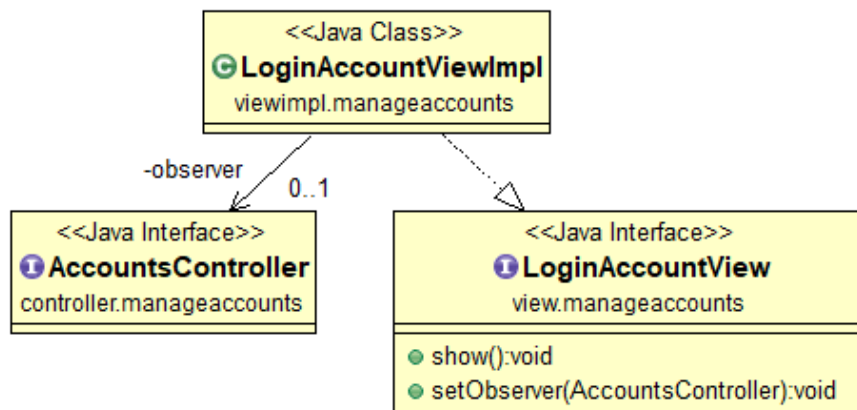


A fronte di un'accurata analisi, ho deciso di creare una sola entità Account, invece che più differenti, in quanto in questo caso sono presenti le stesse caratteristiche descrittive. Questo infatti eviterebbe l'inutile proliferare di codice uguale e soddisferebbe in questo modo anche il principio DRY (Dont' repeat yourself). Infatti un Account qualsiasi è composto da un Nome, un Cognome, un Username e infine una Password.

Tuttavia, nonostante questo, è giusto far notare, che l'utilizzo di questo software è stato pensato per due diversi contesti di impiego, ossia amministrazione e personale, ognuno dei quali è legato a proprie specifiche e permessi. Per questo motivo, per permettere che questo possa accadere, ho dovuto implementare un altro aspetto, oltre alle caratteristiche generali, che differenziasse le due tipologie degli account presenti.

Per rendere possibile questo, ho infatti utilizzato una enum chiamata TypeAccount che stabilisce le due tipologie degli account: "Administrator" e "Operator" e che porta a una non modifica delle tipologie.

Login



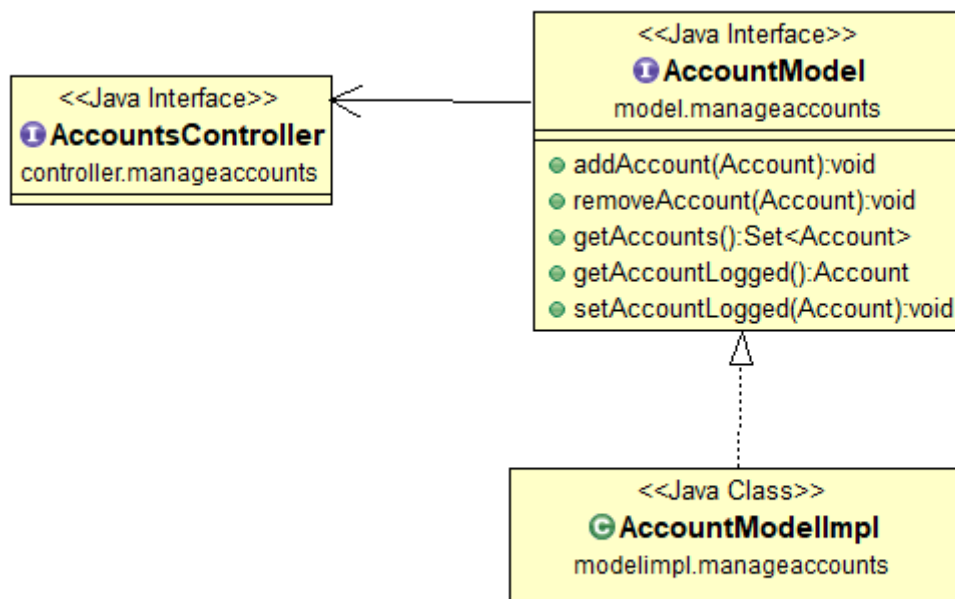
Il primo passo per fare in modo che l'intera applicazione si avvii, è quello di permettere una corretta autenticazione dell'utente definito per poi successivamente mostrare il menù con i permessi di cui l'utente appena autenticato dispone.

Per questo motivo, infatti, all'avvio dell'applicativo, verrà mostrata la schermata di login chiamata **LoginAccountView** in cui si dovrà inserire l'username e la password di cui si dispone.

Per fare in modo che questo reporting avvenga, è stato utilizzato il pattern Observer in modo tale da consentire correttamente la comunicazione uno a molti. In questo caso possiamo infatti notare come l'interfaccia grafica **LoginAccountView** sia Observable e le istanze di input, che in questo caso è svolto dal Controller specificato per la gestione degli Accounts chiamato **AccountController**, sono Observer.

Come è stato descritto anche precedentemente, per il corretto funzionamento dell'applicativo è di fondamentale importanza implementare delle politiche di gestione per ogni entità e funzionalità del nostro dominio per essere correttamente inserite all'interno del sistema.

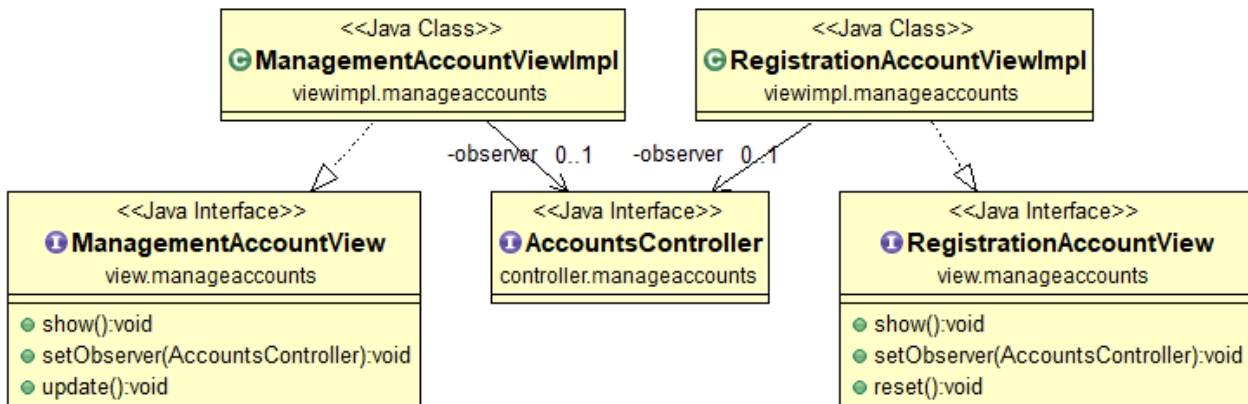
La prima fra tutte che ho dovuto implementare è quella di rendere univoco l'username dell'utente in modo tale da evitare duplicazioni.



Oltre a tutto ciò, essendo presenti due contesti di utilizzo con permessi differenti in quanto a seconda della tipologia dell'utente autenticato saranno resi disponibili operazioni diversificate, è indispensabile in questo momento, il salvataggio dell'account, soprattutto della sua tipologia.

Per fare ciò, ho infatti deciso di salvare l'Account loggato, in modo tale da renderlo disponibile e accessibile al momento dell'avvio del Menù. Questo avviene grazie alla variabile `AccountLogged` che viene presa e settata nel model specifico per la gestione degli account chiamato **AccountModel** grazie agli appositi metodi `get` e `set`, che permettono al **AccountController** di averne sempre accesso e di poterlo rendere disponibile al Menù.

Gestione Account



In seguito all'accesso, l'utente amministratore si troverà all'interno del Menù in cui verranno mostrate le varie sezioni, le quali ricoprono aspetti differenti dell'applicativo.

La sezione di cui, successivamente al Login, mi sono occupata è stata quella inerente alla gestione degli account ossia a tutto il coordinamento, amministrazione, aggiunta ed eliminazione di ogni singolo account presente all'interno dell'applicativo.

Ho reso possibile tutto ciò grazie alla schermata iniziale chiamata **ManagementAccountView** in cui verrà mostrata una tabella di tutti gli account registrati fino a quel momento. Grazie all'utilizzo di questa tabella, la visualizzazione sarà resa più intuitiva e semplificata in quanto verranno mostrati i dati relativi all'account come l'Username, il Nome, Il Cognome e la tipologia dell'account, tutto in una singola riga.

Tutto ciò è possibile grazie all'utilizzo della libreria GSON che con l'implementazione di un metodo all'interno del **AccountController**, permette la lettura e visualizzazione di tutti gli account presenti all'interno del file **Account**.

Registrazione ed eliminazione Account

In **ManagementAccountView**, l'utente amministratore potrà effettuare due operazioni principali oltre alla semplice visualizzazione degli account:

- L'eliminazione di un account già esistente, sia della tipologia amministratore che operatore.
- La registrazione di un nuovo account, anch'esso di entrambe le tipologie.

Nel primo caso la view rende tutto molto intuitivo in quanto basta semplicemente selezionare la riga in cui è presente l'account che si desidera eliminare, per poi successivamente selezionare il bottone relativo all'eliminazione. Tutto ciò porterà all'effettiva cancellazione dell'account grazie al **AccountsController** che comunica con l'**AccountModel** ed effettua la modifica, per poi comunicarlo a sua volta al controller.

Questo viene subito mostrato nell'interfaccia grafica `ManagementAccountView` grazie anche all'uso del pattern `Observer` in cui possiamo notare come la view sia `Observable` e il lavoro svolto dal `AccountController`, sia l'`Observer`.

Per quanto riguarda questa prima operazione però, per evitare e prevenire errori di ogni genere, ho dovuto aggiungere vari controlli.

Il primo consiste nell'evitare di dare la possibilità di cancellare il proprio account, in quanto essendo quello loggato, non potrebbe più rientrare successivamente. Questo è stato reso possibile grazie alla presenza dell'account loggato in quel momento salvato nel `AccountModel` che comunicandolo all'`AccountController` riesce ad informarlo ed evitare attraverso un semplice controllo nella view `ManagementAccountView` che questo avvenga.

Un altro controllo invece consiste nell'evitare di dare all'account presente in quel momento, la possibilità di cancellare l'ultimo account di tipo `Administrator` in quanto se questo fosse possibile, successivamente nessun utente riuscirebbe ad accedere a questa zona di gestione account rendendo così il tutto inutilizzabile. Questo è stato reso possibile grazie all'utilizzo di un filtro che prende in considerazione gli account del tipo `Administrator`, trovandone così alla prima corrispondenza con quello presente all'interno della tabella.

In caso di mancato selezionamento, verrà mostrato un avviso senza compromettere nessun account presente.

Nel secondo caso invece, ossia l'aggiunta di un nuovo account, viene mostrata una nuova interfaccia chiamata `RegistrationAccountView`, in cui vengono richiesti i dati base per una corretta registrazione dell'account come il Nome, Cognome, Username, Tipologia e Password.

Come anche nel primo caso, anche per questa operazione, per far fronte alle varie problematiche che si potrebbero verificare all'aggiunta di un nuovo utente, sono stati implementati vari controlli. I primi fra tutti fanno riferimento al puro vero e proprio inserimento dei dati da parte dell'utente. Infatti consistono nell'evitare di lasciare campi fondamentali vuoti, come il Nome, il Cognome o l'Username, oppure di evitare di non dare la possibilità all'utente di poter inserire numeri all'interno del Nome e del Cognome.

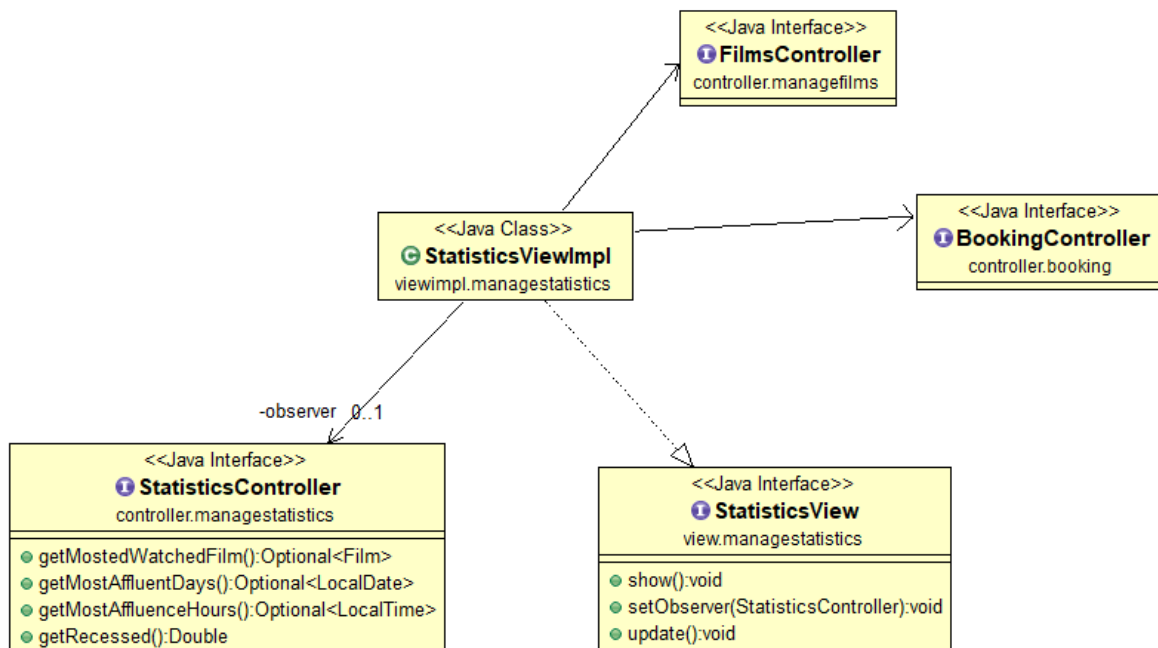
Successivamente è presente un controllo già fondamentale per la prima parte del Login, in cui l'username inserito dovrà essere univoco.

Un altro controllo di fondamentale importanza è definito grazie ad un inserimento duplicato della password al momento della creazione in quanto in questo modo sarà possibile verificare la corrispondenza tra i due inserimenti, per agevolare l'utente, e se la corrispondenza avverrà, allora quell'inserimento potrà essere registrato come password.

Come già descritto precedentemente per la tabella della view `ManagementAccountView`, anche in questo caso tutto ciò è possibile grazie all'utilizzo della libreria `GSON` che con l'implementazione di un metodo all'interno del `AccountController`, permette la scrittura e visualizzazione di tutti gli account presenti all'interno del file `Account`.

Come è possibile notare soprattutto in questo caso, ho preso la decisione di creare più classi e interfacce grafiche per rendere il codice il più intuitivo possibile all'utente, rispettando anche il principio `SRP` (`Single Responsibility Principle`) che raccomanda di evitare di costruire classi che gestiscono responsabilità diverse.

Statistiche base



A seguito della aggiunta e programmazione di uno o più Film e successivamente della vendita di biglietti ai clienti, l'amministratore potrà tranquillamente visionare le statistiche base presenti relative a tutto il gestionale. Questo sarà possibile accedervi grazie all'interfaccia grafica **StatisticsView** nella quali sono presenti tutte le statistiche assieme.

Le statistiche disponibili sono in totale quattro, ma è sempre possibile aggiungerne molteplici grazie all'utilizzo di una tabella presente all'interno di **StatisticsView** in cui la visualizzazione di quest'ultime sarà semplificata.

- Film più visto
- Data più affluente
- Ora più affluente
- Incasso totale

Statistiche: Film più visto

La prima statistica base che incontriamo a sinistra nell'interfaccia grafica, è il film presente al Cinema più visto con la relativa copertina, se questa presente.

Per fare in modo che questo possa essere calcolato, devo tenere in considerazioni vari aspetti.

Per poter calcolare questa prima statistica, ho bisogno di avere accesso ad altri dati che non sono di mia competenza in quanto descrivono aspetti di altre entità principali quali Film e vendita dei biglietti.

Infatti per superare questo ostacolo, ho utilizzato il pattern Observer avendo

così **StatisticsView** come **Observable** e come **Observer** **FilmsController** e **BookingController**.

Questo perché per calcolare il film più visto, mi è indispensabile conoscere come prima cosa il numero massimo della vendita dei biglietti e successivamente sapere a quale film quella vendita corrisponde. Per questo motivo, mettendo così in comunicazione i tre controller fra loro, grazie alla

creazione di un filtro presente nel `StatisticsController`, posso ricavare il dato che mi serve e successivamente mostrarlo alla view che mi interessa.

Tuttavia, tutto ciò non basta in quanto bisogna fare attenzione al fatto che quando un film qualsiasi sarà eliminato dall'utente, conseguentemente verranno rimosse tutte le relative programmazioni e i biglietti venduti ad esso riferiti. Questo aspetto è molto importante da considerare in quanto se ipoteticamente l'utente amministratore decidesse di eliminare ogni film e programmazione, a questo punto non esisterebbe nessun film che potrebbe ricoprire il ruolo in questa statistica.

Per oltrepassare ciò, ho infatti adottato un `Optional` che accetta anche la non presenza di un `Film` risolvendo in questo modo il problema.

Statistiche: Data e ore più affluente

La seconda e terza statistica sono molto simili ma prendono in considerazione due dati differenti. La prima tra le due il giorno, la seconda l'orario.

Anche in questo caso per poter calcolare queste statistiche, ho avuto il bisogno di accedere ad altri dati che descrivevano aspetti delle altre entità quali sempre `Film` e vendita dei biglietti.

Come già superato precedentemente, anche in questo caso, ho deciso di adottare il pattern `Observer` avendo così `StatisticsView` come `Observable` e come `Observer` `FilmsController` e `BookingController`.

Entrambi questi due controller mi sono indispensabili per poter calcolare la statistica in quanto come primo passo ho bisogno di calcolare il numero della vendita dei biglietti riferiti ad una data soltanto per poi infine confrontare e trovare in quale data c'è stata una vendita maggiore. Grazie all'utilizzo del pattern e della creazione di un nuovo filtro presente nel `StatisticsController`, sono riuscita a ricavare i dati necessari per poter permettere la visualizzazione del risultato successivamente nella tabella presente in `StatisticsView`.

Lo stesso ragionamento viene applicato alla seconda tra queste due statistiche, con l'unica differenza che nel filtro invece di tenere in considerazione la data, si prede in considerazione l'orario di programmazione, mantenendo comunque la comunicazione fra i vari controller.

Come nella statistica del film più visto, anche in questo caso, bisogna però tenere in considerazione il fatto che quando un film sarà eliminato, conseguentemente verranno rimosse tutte le relative programmazioni e i biglietti venduti. Per far fronte a questo problema, ho nuovamente adottato un `Optional` che accetta anche la non presenza di una data o un orario risolvendo in questo modo il problema.

Statistiche: Incasso totale

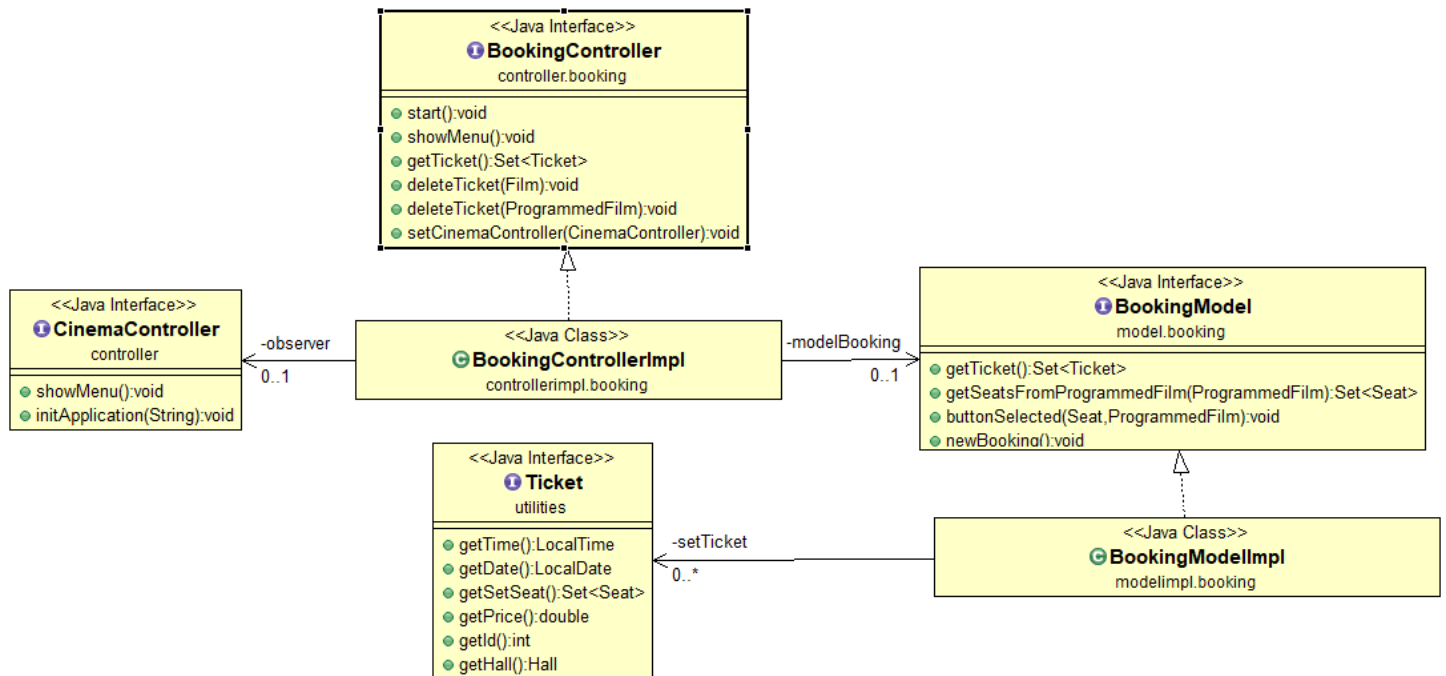
Questa ultima statistica rappresenta il totale guadagno che il gestionale incassa grazie alla vendita dei biglietti a seconda del prezzo prestabilito dall'utente per una determinata programmazione.

Come già spiegato precedentemente, anche in questo caso, ho deciso di adottare il pattern `Observer` avendo così `StatisticsView` come `Observable` e come `Observer` solo `BookingController` in quanto per calcolare l'incasso ho solamente bisogno di accedere al prezzo della prenotazione dei posti venduti.

Tutte le mie implementazioni di tutte le classi sono sempre state create rispettando solidamente il principio cardine DIP (Dependency Inversion Principle) in cui ogni classe dipende solamente da interfacce e non dalle loro implementazioni

2.2.2 Lorenzo Sansone

Per gestire le prenotazioni dei posti per un determinato film in programma si è deciso di creare un



controller specializzato che gestisca questa funzionalità.

Il controller in questione è **BookingControllerImpl**, esso implementa diverse interfacce (come si può notare anche dalla figura dopo) ma la principale è **BookingController**. Questa è la sua interfaccia generale che permette di non dare libero accesso all'esterno a tutti i metodi implementati ma solo a quelli che possono interessare ad altri componenti, per esempio il controller principale **CinemaController** utilizzando questa interfaccia non potrà accedere a metodi che riguardano principalmente la gestione delle interfacce grafiche.

In breve **BookingControllerImpl** gestirà tutte le view destinate alle prenotazioni e avrà un model dedicato per mantenere traccia dello stato dell'applicazione per quanto riguarda la vendita dei biglietti.

Si utilizza il pattern Observer per notificare al **CinemaController** che si vuole ritornare al menu principale ogni qualvolta l'utente lo richieda. L'observer è **CinemaController** invece l'observable è **BookingControllerImpl**.

Il controller in questione provvederà ad aggiornare i file per il salvataggio dei dati permanenti ogni qualvolta si eseguirà una prenotazione. Mentre la lettura viene fatta solo all'atto dell'istanziamento.

Il **BookingController** ha un'istanza di **BookingModelImpl** ed esso è responsabile per il mantenimento dello stato corrente dell'applicazione per quanto riguarda la vendita dei biglietti.

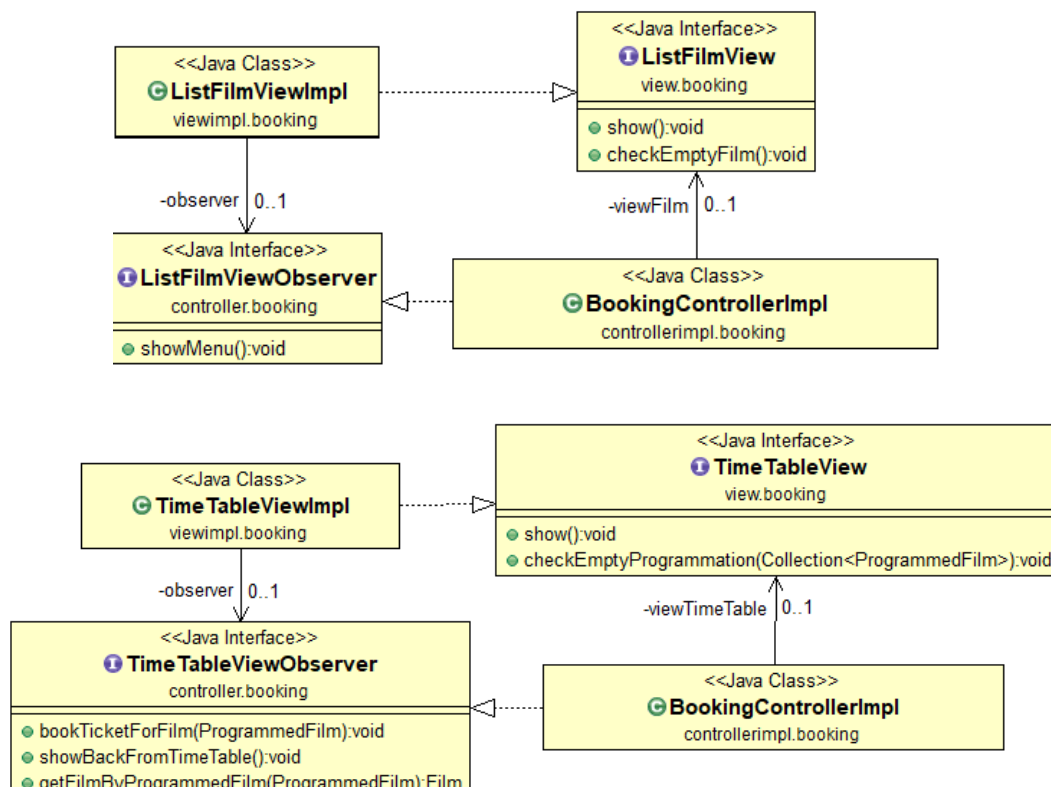
In particolare sono stati implementati metodi per la cancellazione e aggiunta di essi. Più precisamente gestisce istanze di Ticket, una delle quali indica tutti i biglietti venduti per un particolare film in programma.

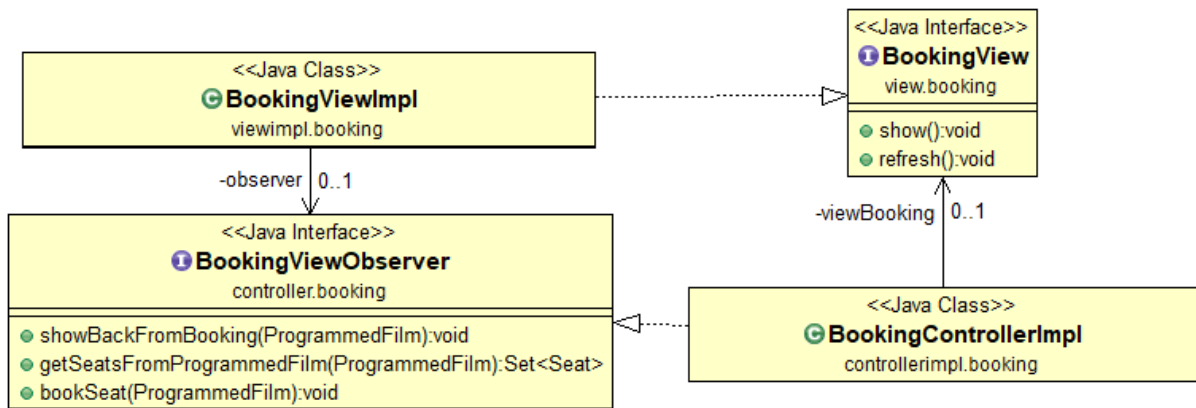
In TicketImpl uno degli attributi più importanti da memorizzare è l'id di un film. Ho evitato di mettere il riferimento diretto all'oggetto Film perché non sono utilizzati pesantemente tutti gli altri metodi della sua interfaccia.

Il metodo "equals" di TicketImpl utilizza l'id, la data, l'orario e la sala di proiezione in quanto non potranno mai essere proiettati due film contemporaneamente nella stessa sala allo stesso tempo. Di conseguenza riesco ad assegnare i posti prenotati per una determinata programmazione in modo sicuro.

Il BookingModelImpl mantiene anche i posti selezionati di una sala in fase di prenotazione e provvederà a schedarli quando il controller verrà notificato dalla view. Nella fase di registrazione se il model scopre che per quella programmazione sono stati già prenotati dei sedili, si limiterà ad aggiornare l'istanza esistente aggiungendo i posti a quelli già presenti.

Ovviamente l'accesso al BookingModelImpl è permesso solo al BookingControllerImpl quindi tutte le modifiche devono essere coordinate da quest'ultimo.

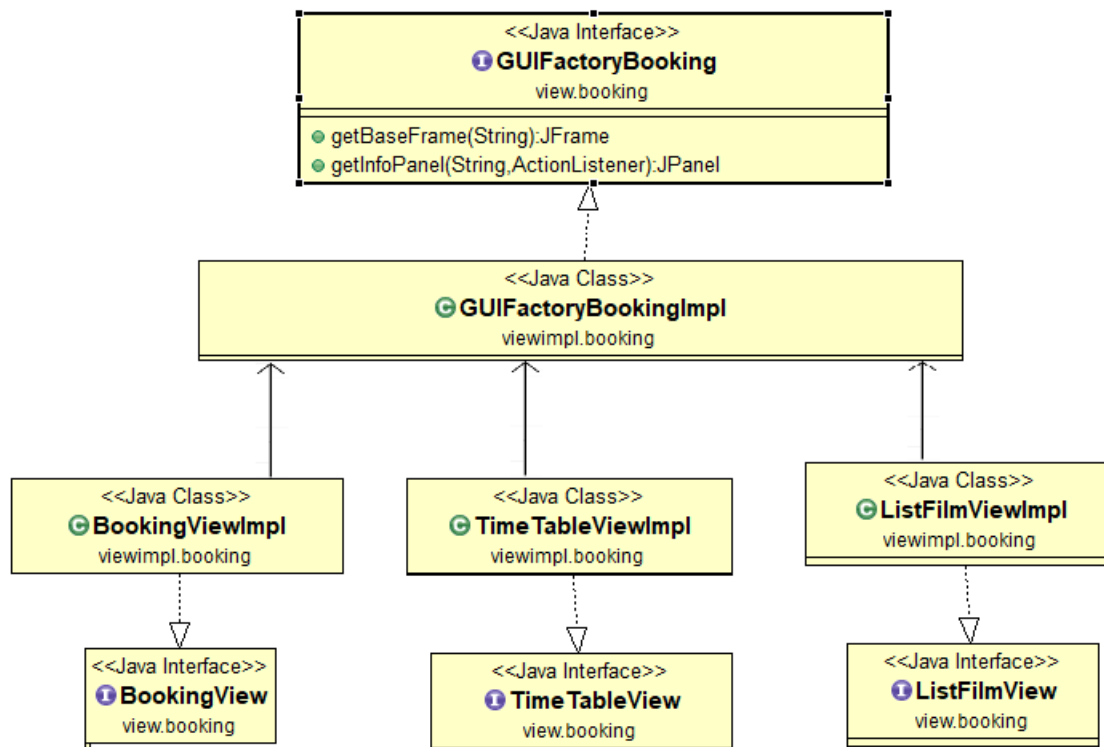




Le restanti interfacce **BookingViewObserver**, **ListFilmViewObserver**, **TimeTableViewObserver** sono state create per utilizzare il pattern Observer, in tal modo la view potrà notificare al **BookingControllerImpl** gli input dell'utente ed agire di conseguenza nel modo più opportuno. Tutte queste interfacce sono collegate ad una view precisa in modo tale che ogni GUI possa utilizzare solo determinati metodi del controller, limitando l'accesso ai metodi dell'oggetto cercando così di seguire il principio ISP (Interface Segregation Principle)

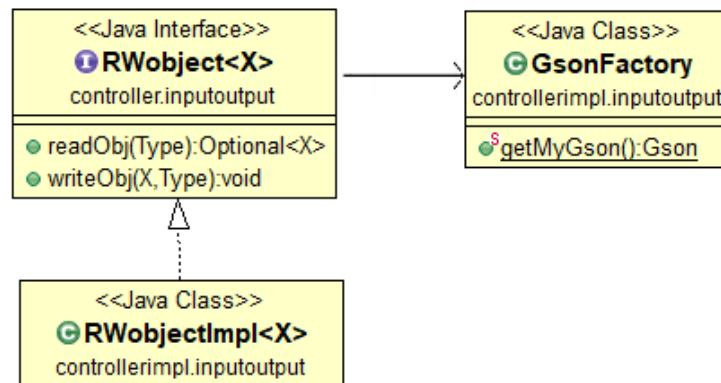
Dunque **ListFilmView**, **TimeTableView** e **BookingView** sono l'observable e **BookingControllerImpl** in questo caso l'observer.

La prenotazione dei posti per un film è dunque divisa in tre step per rendere più user-friendly possibile il processo. In particolare **ListFilmView** viene scelto il film per cui si vuole procedere, in seguito attraverso **TimeTableView** si sceglie la programmazione precisa per il film scelto precedentemente ed infine con **BookingView** si selezionano e prenotano i posti.



Tra le varie View ci sono dei componenti in comune o impostazioni dei componenti grafici che possono condividere. Per risolvere ciò si è provveduto a costruire una factory. L'oggetto **GUIFactoryBookingImpl** fornisce metodi per la costruzione di oggetti grafici.

Lettura e scrittura



Una delle parti fondamentali del nostro gestionale è il mantenimento dei dati che si protrae dopo lo spegnimento della macchina quindi si è optato per la scrittura e lettura su file per rendere permanenti alcuni dati che sono fondamentali per la continuità del gestionale.

Dato che all’inizio non erano definitivi e ben delineati gli oggetti da salvare si è creata una serie di classi che permettevano di leggere e scrivere tutti gli oggetti aggiungendo poche righe di codice.

In particolare l’interfaccia parametrica **RObject** specifica due soli metodi, uno per leggere e uno per scrivere, prendendo come variabile d’ingresso un “Type” che aiuta a capire a runtime che cosa bisogna leggere o scrivere evitando durante l’esecuzione fenomeni di type erasure.

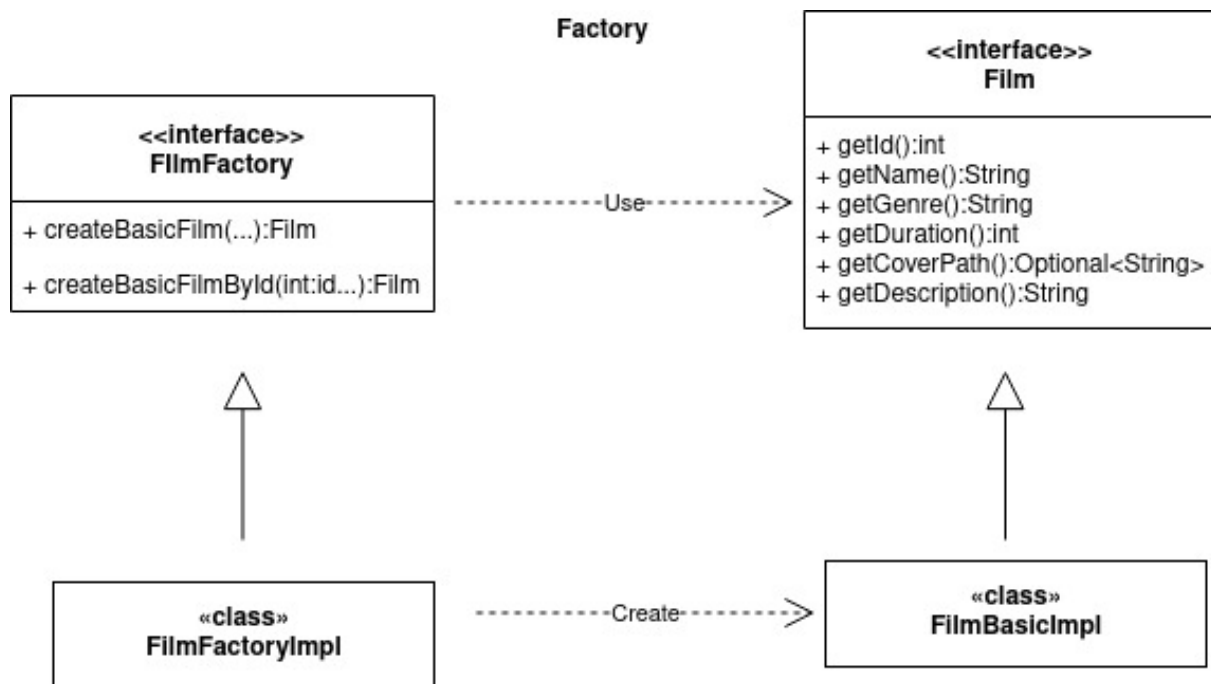
L’implementazione **RObjectImpl** implementa **RObject** ed utilizza una factory statica per costruire un oggetto **Gson** con gli adapter necessari per riuscire a scrivere/leggere le classi che dovranno essere usate.

L’implementazione **RObjectImpl** potrà così essere istanziata ed utilizzata dai controller per salvare i dati fondamentali specificando il loro tipo.

2.2.3 Alessandro Zirondelli

Il mio compito all'interno del Team , per la realizzazione di questo sistema, è stato quello di implementare la gestione dei film, della loro programmazione e dell'avvio/inizializzazione dell'applicazione.

Film



La prima cosa che ho deciso di fare è stata quella di implementare il concetto di film.

Ad esso ho voluto attribuire univocamente un id , per identificarlo e distinguerlo , ed alcune proprietà/caratteristiche che saranno inserite nel momento della creazione.

La fase di creazione di questa entità è stata delegata ad una apposita factory per disaccoppiare le responsabilità e l'effettiva implementazione del film.

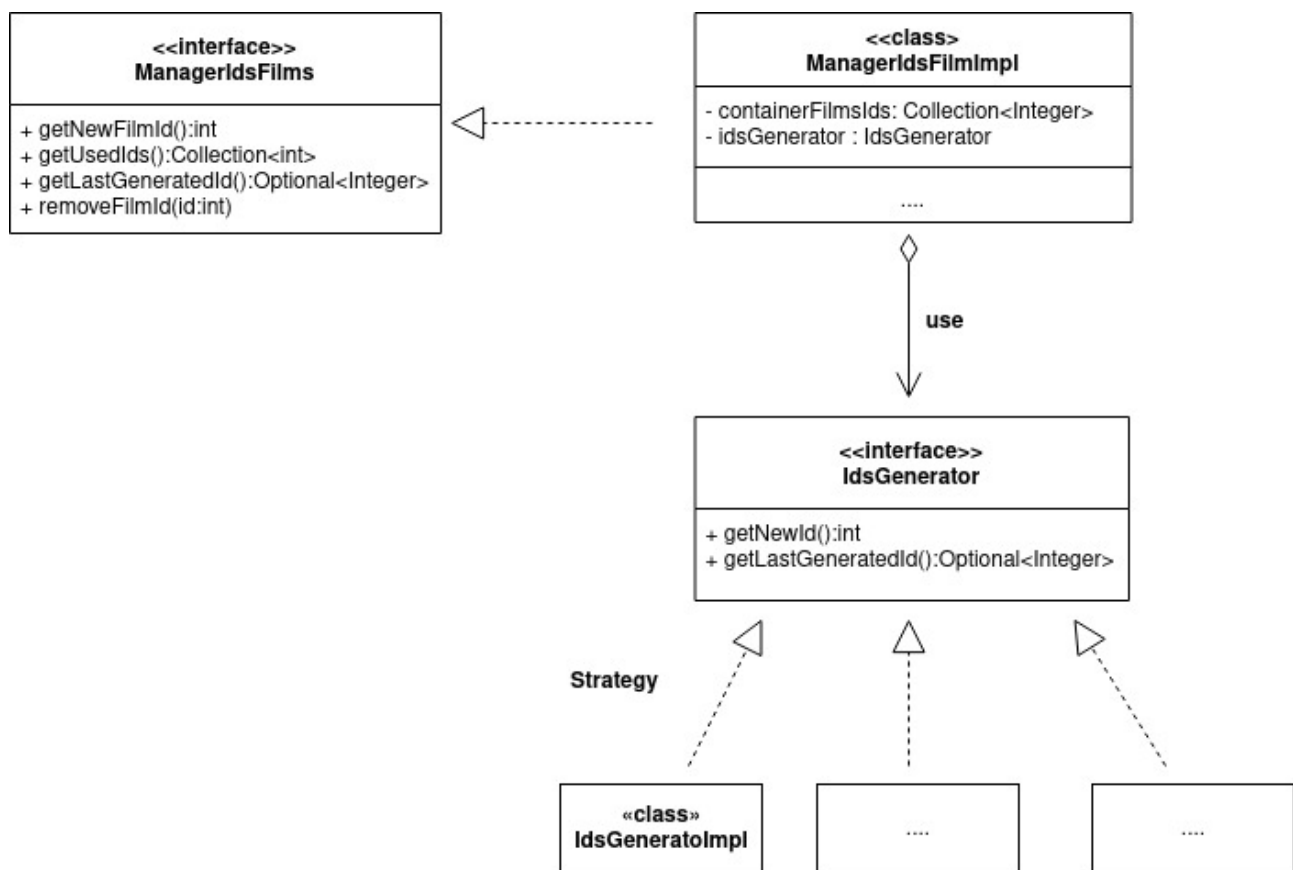
Inoltre è stato necessario per rendere più chiaro ed intuibile le due diverse metodologie di creazione dell'oggetto , delegare la generazione dell'id ad un'altra entità (che vedremo in seguito) oppure fornendolo direttamente.

L'entità Film è stata pensata per essere immutabile, in modo che non possa essere modificato in alcun caso dall'esterno il suo stato attuale. Questa scelta è stata fatta per evitare la perdita o la modifica di dati che poi andranno scritti in modo permanente su disco.

Sapendo che lo stato corrente di un particolare oggetto rimane invariato ed è sempre noto , ha favorito e potrà farlo in futuro lo sviluppo di test, dato che determinate asserzioni verranno sempre soddisfatte.

Oltre all’inserimento di un film , ho dovuto implementare anche una eventuale sua modifica, per cui ho riflettuto sulla scelta fatta e ho concluso che non sarebbe stato comunque un problema dover ricreare ad ogni singola modifica l’oggetto con proprietà differenti in quanto sarebbe stata un operazione semplice e leggera. Da qui nasce il motivo per il quale è stato implementata la creazione del film fornendo già uno specifico ID. Questo è necessario in quanto i film programmati hanno un riferimento allo specifico film, ma questo lo vedremo in seguito.

ManagerIdsFilms

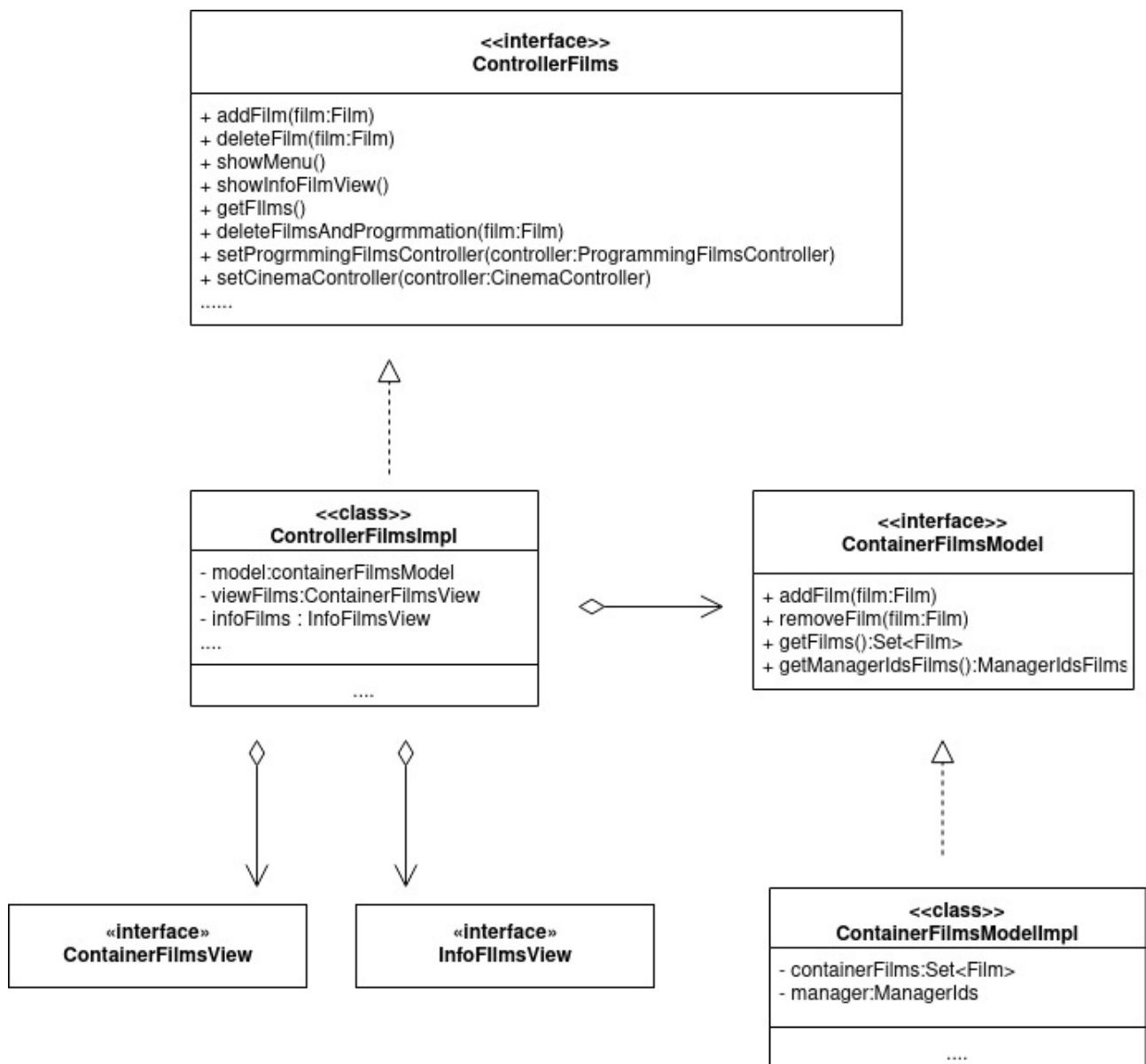


Come anticipato precedentemente la gestione degli ID relativi ad ogni film è assegnata ad una specifica entità denominata **ManagerIdsFilms**. Essa possiede un contenitore di ID utilizzati e un riferimento a un generatore , **IdsGenerator**. Questo si occupa di generare effettivamente il nuovo ID da assegnare al film. Per il calcolo dell’ID, può utilizzare differenti strategie di generazione in base a quella che viene implementata. Ho deciso quindi di creare una strategia di tipo progressivo

(IdsGeneratorImpl) lasciando però la possibilità in futuro di crearne altre in base a possibili ed eventuali esigenze.

Da ciò si può dedurre che ho utilizzato il pattern strategy, dove la strategia è rappresentata da IdsGenerator.

Gestione Film



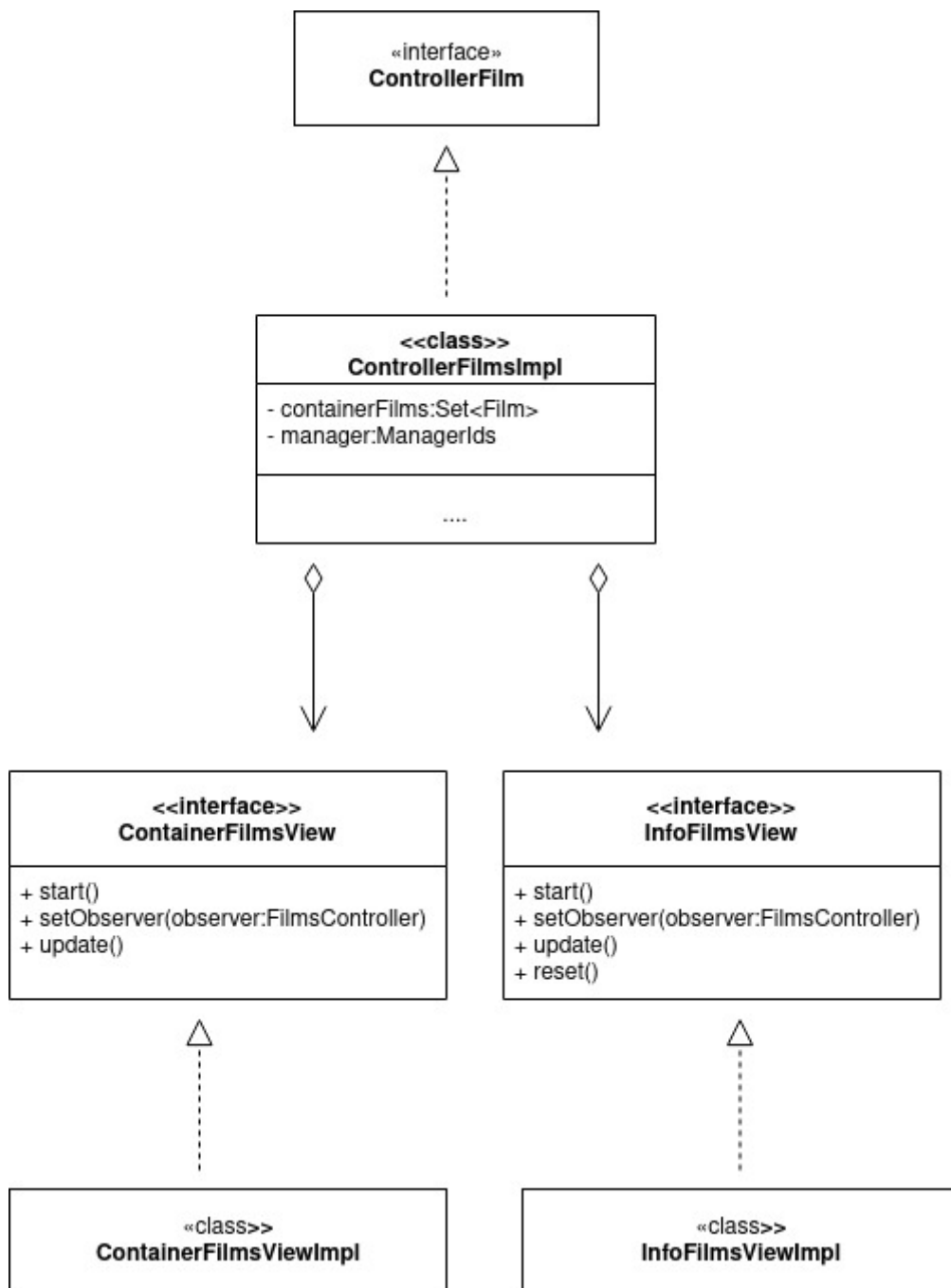
Il salvataggio corrente dello stato dell'applicazione relativo ai film, è possibile grazie ad uno specifico model, denominato ContainerFilms. Esso permette di effettuare le varie operazioni che possono essere apportate sui films.

L'interazione con l'utente avviene tramite due view che si occupano della visualizzazione di tutti i film (ContainerFilmsView) e dell'inserimento/modifica/eliminazione (InfoFilmsView).

La comunicazione tra queste e il model avviene tramite un controller specifico, FilmsController. Quest'ultimo è observer delle schermate observable appena citate, ed anche del controller principale, ControllerCinema. Ho utilizzato quindi un pattern observer per notificare i cambiamenti al model attraverso questo controller intermediario.

Quest'ultimo è anche il responsabile, una volta aggiornato il model, di memorizzare in modo persistente i nuovi cambiamenti su file.

View Films

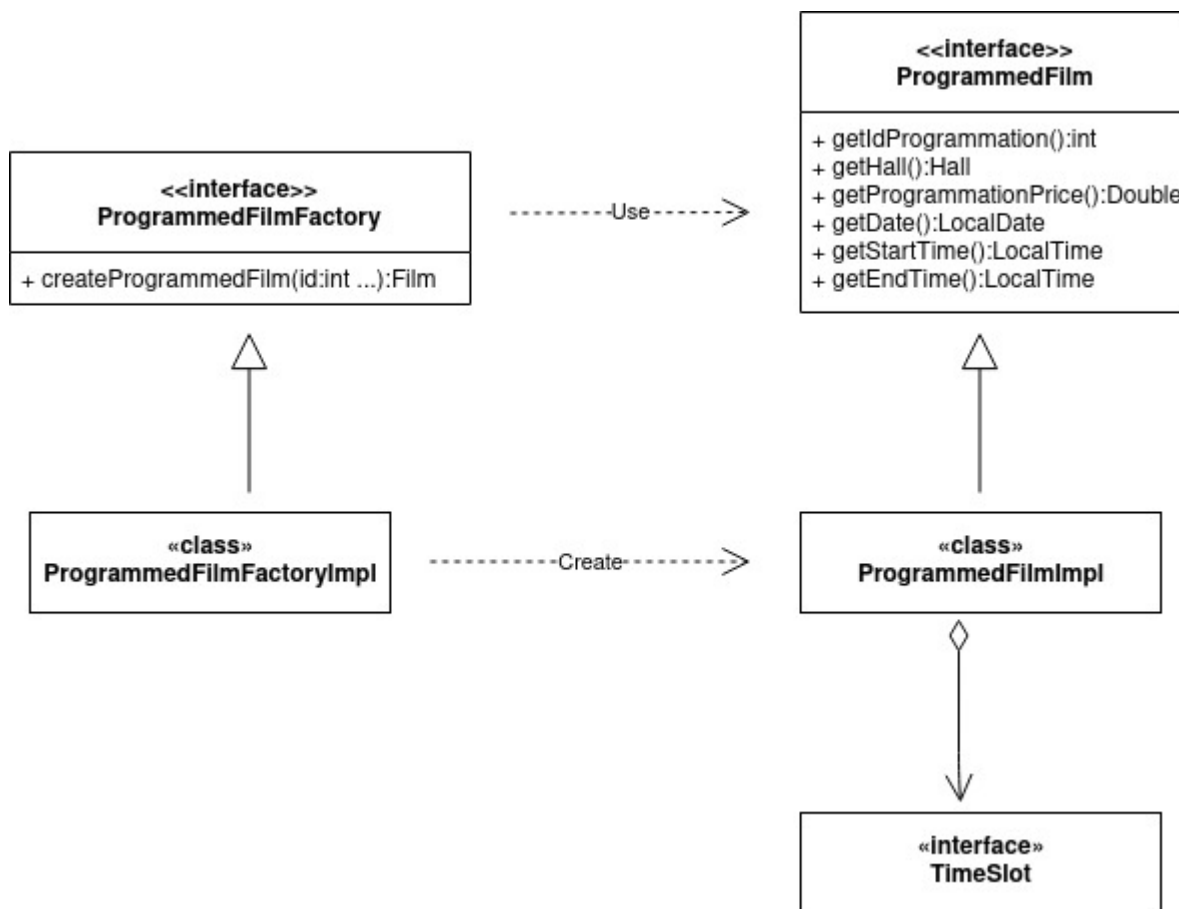


La correttezza nell' inserimento dei dati del film è assicurata tramite una gestione delle eccezioni, che notifica l'utente di un'immissione non valida tramite GUI.

Attraverso il metodo `setObserver` il controller diventa l'observer della view e notifica i cambiamenti al model che provvede ad un aggiornamento di stato.

Dopodichè le view possono aggiornarsi tramite `update`, reperendo tramite controller le nuove informazioni dal model.

Film programmato



Per quanto riguarda la programmazione dei film ho proceduto con l'implementazione dell'entità **ProgrammedFilm**, la quale contiene le informazioni necessarie per identificarla e per capire a quale film si sta riferendo, tramite un ID reference. Ho deciso di fare questa scelta, evitando di usare una object reference a **Film** per avere inanzitutto meno dipendenze, per rendere più semplice ed intuitiva la logica che sta dietro alla programmazione e più leggera/performante la scrittura su file con GSON. Inoltre perchè non mi sarebbero servite tutte le informazioni contenute nel film. Questa decisione è stata presa anche per ottemperare al principio KISS(Keep It Simple Stupid).

Per il salvataggio della fascia oraria di schedulazione del film, viene utilizzata l'entità **TimeSlot**, che memorizza un'orario di inizio e di fine.

La creazione del film programmato avviene anche qui tramite factory. Tutti i dati memorizzati inerenti alle programmazioni vengono salvati all'interno di un model in particolare, **programmedFilmsModel**, che offre le varie operazioni disponibili.

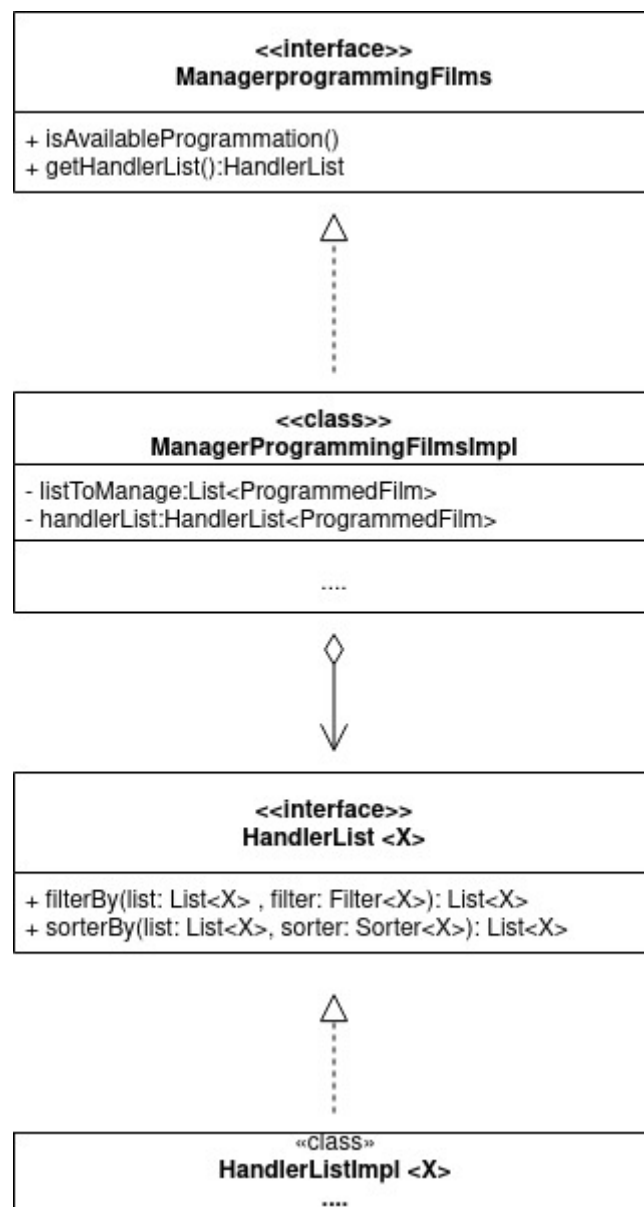
Facendo riferimento all'analisi del dominio precedentemente esposta, è emerso di fondamentale importanza creare dei meccanismi di gestione delle varie operazioni.

In particolare, qui risultano significativi diversi aspetti:

- Navigazione tra le programmazione passate, odierne e future

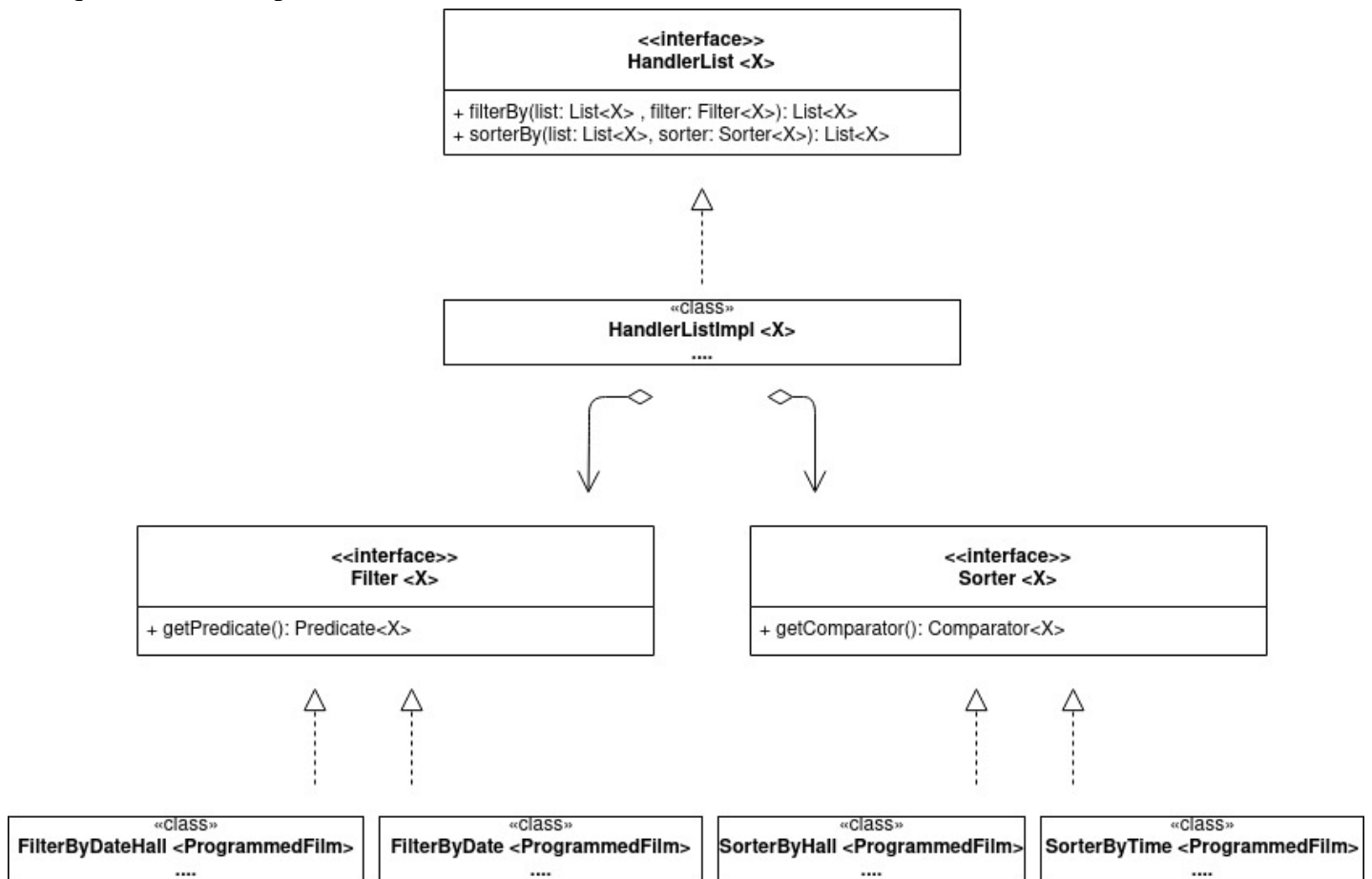
- Aggiunta di una programmazione solo se correttamente verificate e controllata. Il che significa non permettere la schedulazione se gli orari si sovrappongono all'interno della stessa sala e nella medesima data. Oppure impossibilitare la schedulazione di un film nel passato.

Manager Programmazione dei Film



Per questo problema ho deciso di implementare e delegare la responsabilità ad un manager, `ManagerProgrammingFilms`, che si occupa della gestione di tutte le programmazioni, seguendo il principio SRP (Single Responsibility Principle)

Quest'ultimo utilizza un `HandlerList` parametrico che offre le due operazioni di filtraggio e ordinamento degli elementi su una lista. Il motivo che mi ha spinto a creare questa entità parametrica è il riuso e la flessibilità del codice, in modo che possa essere usata anche in futuro per qualche altro scopo.

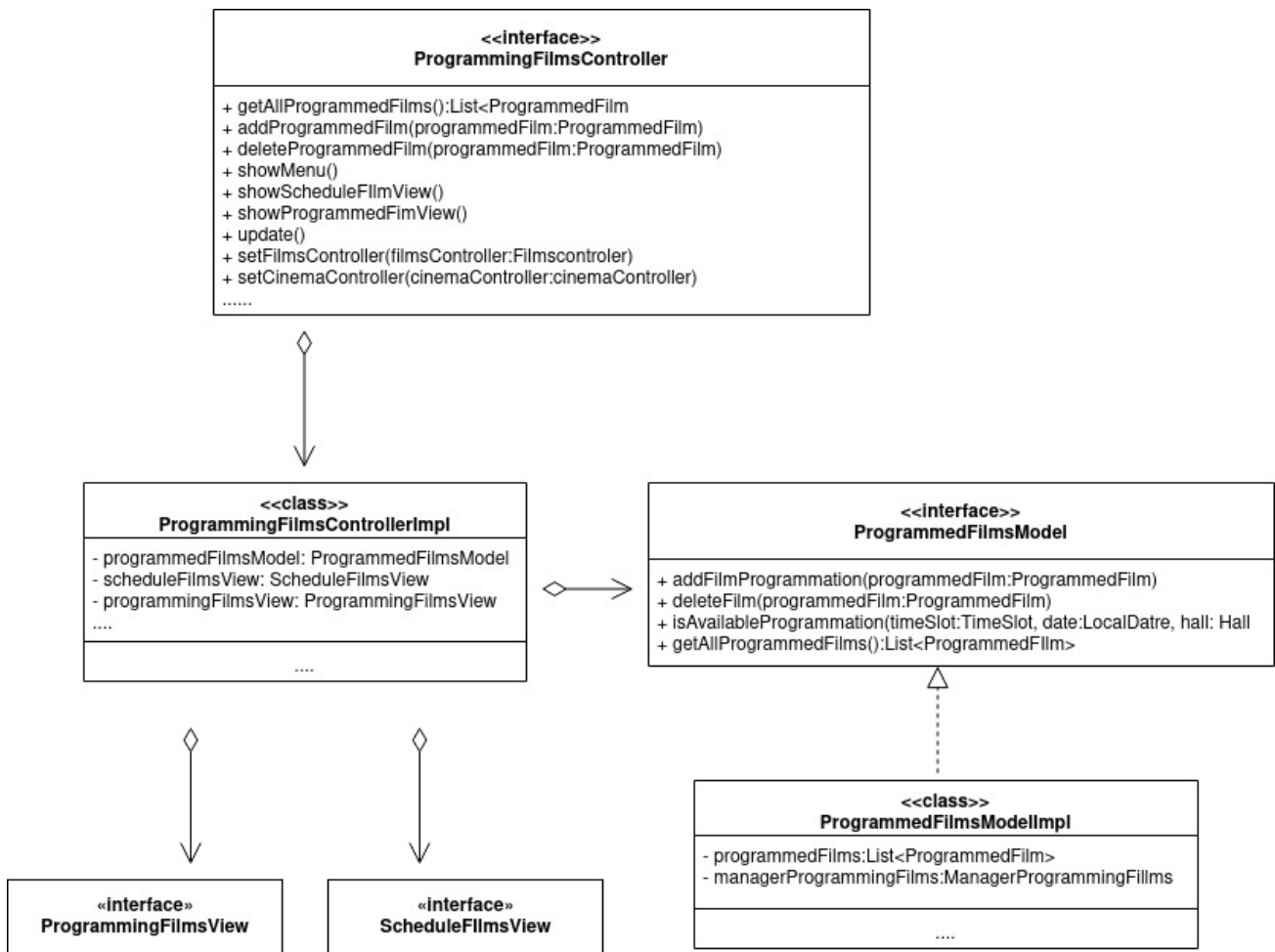


A tal proposito ho deciso di creare anche un'interfaccia `Filtro` e `Sorter`, sempre di tipo parametrico in modo da rivelare maggiormente l'intento di quello che ho deciso di fare, in aggiunta alle motivazioni sopra citate. Tutte le varie implementazioni di queste due entità sono necessarie nella risoluzione della problematica relativa alla correttezza di inserimento della programmazione.

Queste infatti costituiscono essenzialmente la logica di questo controllo.

Se in futuro sarà necessario apportare dei cambiamenti, modificando eventualmente dei filtri, la soluzione sarà semplice: creare una nuova implementazione ottemperando all'interfaccia `Filter` con le operazioni desiderate.

Gestione film programmati

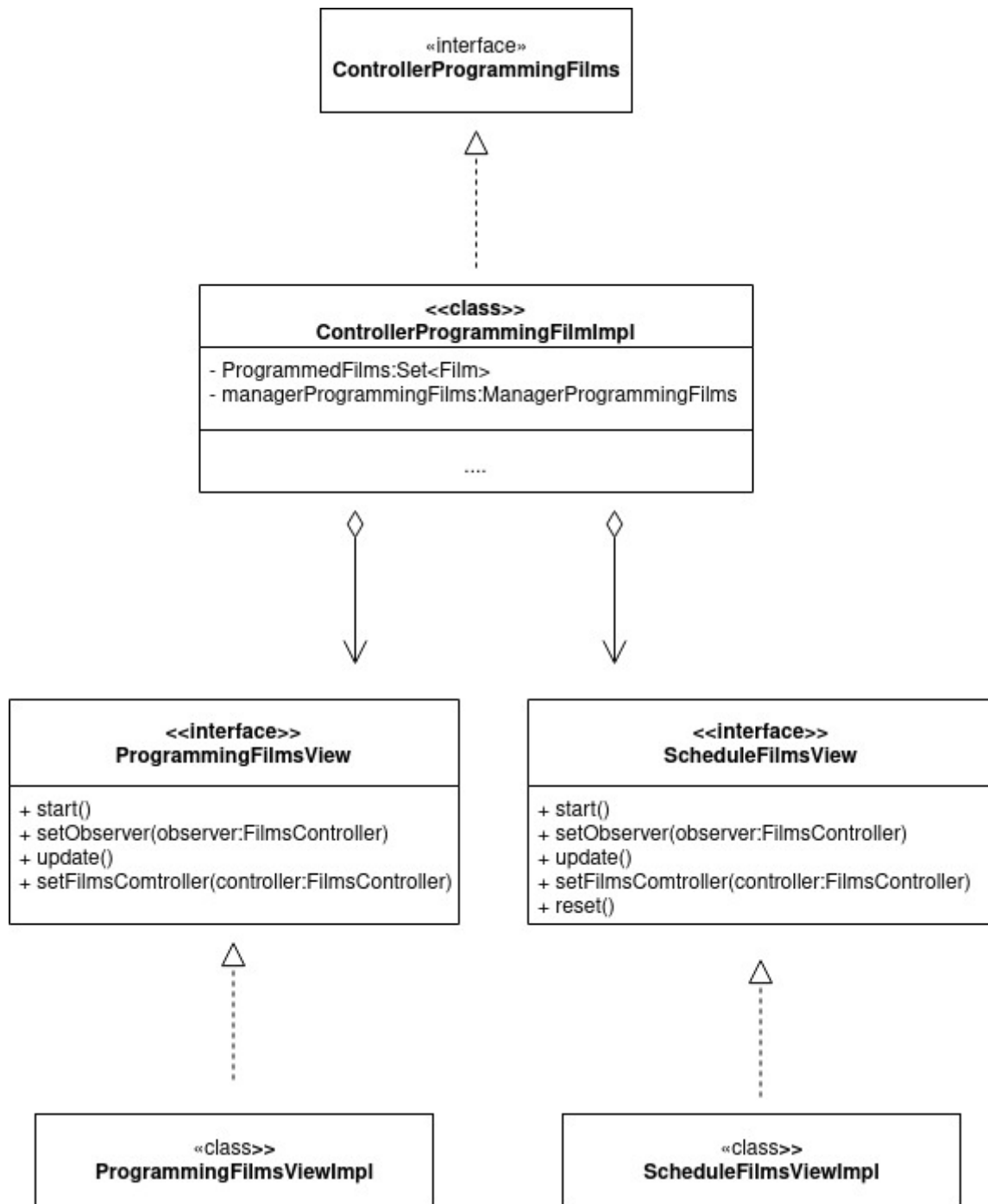


Un requisito richiesto in fase di analisi, è stato quello di dare la possibilità di eliminare delle programmazioni e di conseguenza anche i relativi ticket venduti. Per cui è stato necessario avere un riferimento a `BookingController`, il quale espone le funzionalità per farlo.

Similmente, `filmsController` deve poter rimuovere tutte le programmazioni di quel particolare film nel caso in cui questo venga eliminato. Ciò motiva il fatto di avere un riferimento a questo specifico controller.

L'interazione con l'utente avviene mediante due view, `ProgrammingFilmsView` e `ScheduleFilmsView`, una per la consultazione/eliminazione delle programmazioni ed una per la schedulazione effettiva. Entrambe le view effettuano controlli di inserimento dei dati e lanciano eccezioni in caso ci siano parametri non validi. Di particolare importanza è stata l'eccezione che rappresenta l'impossibilità di schedulare a causa della indisponibilità della sala, in quella fascia oraria e data.

View Film programmati



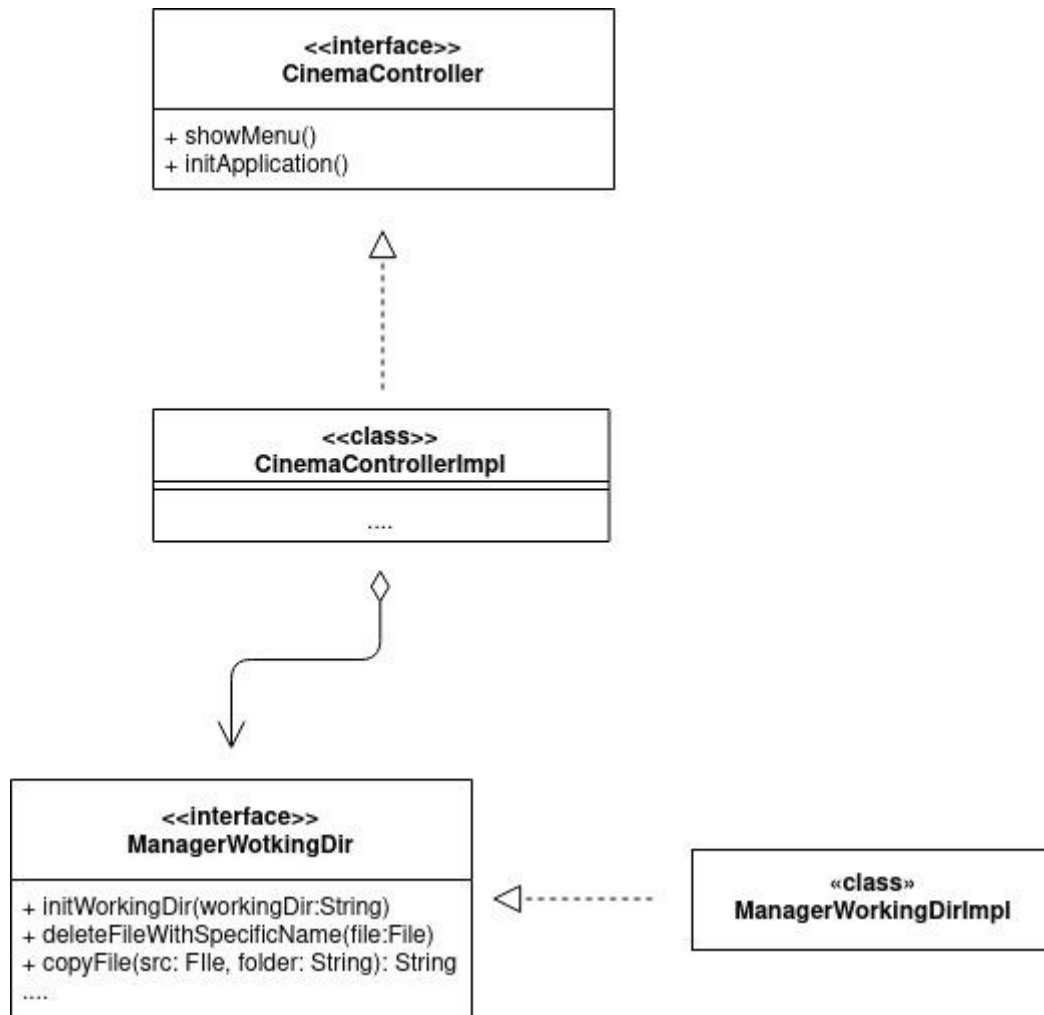
La comunicazione tra queste view e il model avviene anche qui tramite uno specifico controller, **ProgrammingFilmsView**.

Mediante il metodo `setObserver` il controller diventa l'observer delle view observable e notifica i cambiamenti al model che provvede ad un aggiornamento di stato.

Dopodichè le view possono aggiornarsi tramite `update`, reperendo attraverso controller, che si occupa anche della scrittura dei dati su file, nuove informazioni dal model.

Inizializzazione applicazione

L'ultimo mio compito è stato quello di implementare l'inizializzazione del sistema per permettere il salvataggio dei dati su disco. Questo rispetta esattamente un requisito primario esposto in analisi.



Ho deciso di creare una entità **ManagerWorkingDir** che si occupa dell'inizializzazione dello spazio di lavoro e della creazione dei file necessari per il corretto avvio dell'applicazione.

Essa va a creare ed inizializzare un directory nella user home dell'utente, in cui attraverso un apposito controller verranno salvati i file GSON con i dati.

E' stata inoltre creata una classe utilitaria, alla quale il manager si appoggia, all'interno della quale sono stati inseriti tutti i percorsi, **GeneralSettings**.

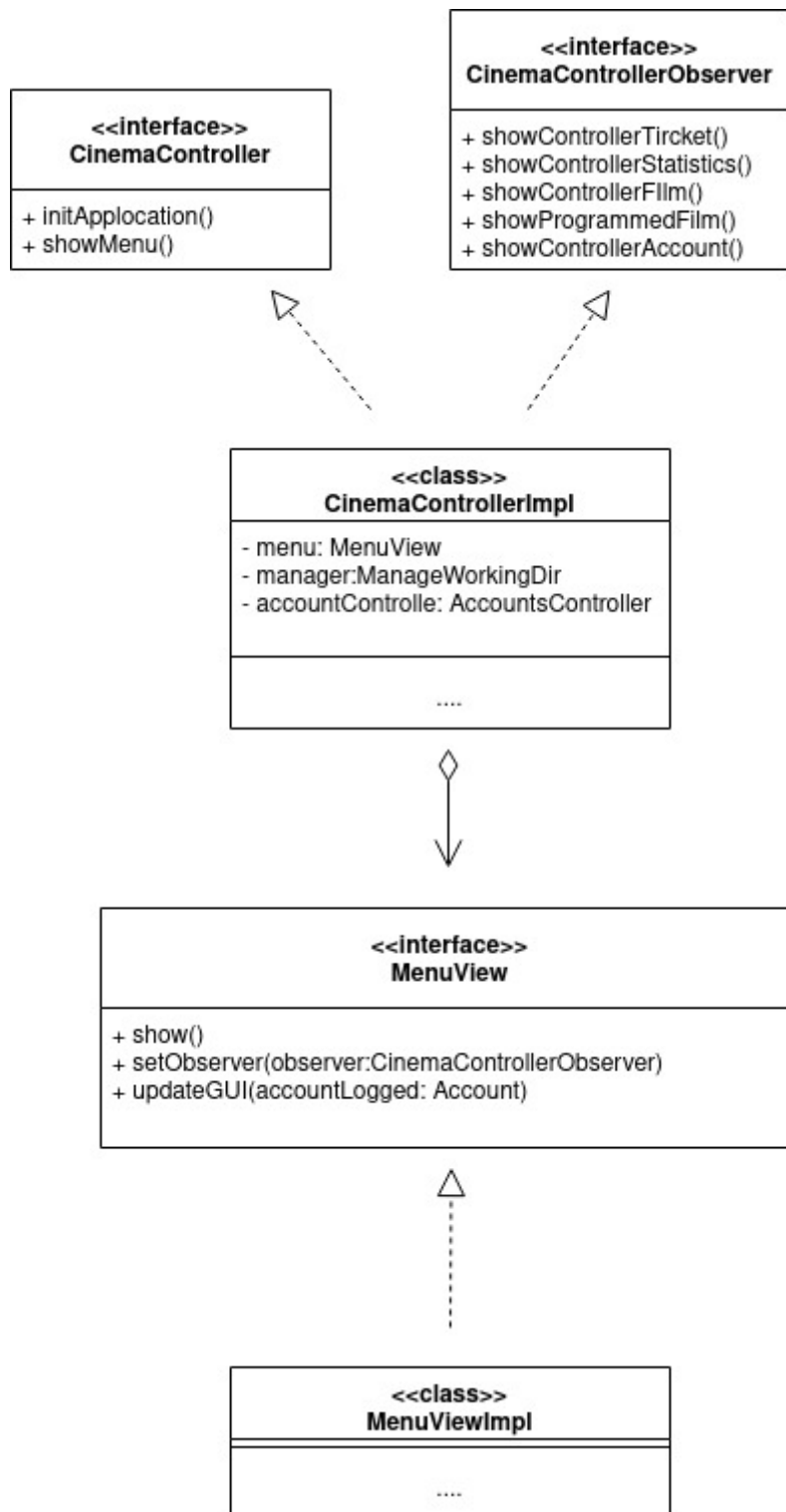
2.2.4 Alessandro Zirondelli e Lorenzo Sansone

Sale

<code><<Enum>></code> Hall
+ HALL_1, HALL_2, HALL_3, HALL_4, HALL_5
+ field: NUM_ROWS
+ field: NUM_COLUMNS

Dato che è stato necessario implementare in breve termine le entità sale abbiamo optato per una classe enum. Siamo completamente consapevoli che impediamo l'estendibilità e riuso del codice violando diversi principi come OCP ma siamo stati costretti per motivi di cui abbiamo discusso nella parte iniziale della "Metodologia di lavoro". Abbiamo supposto quindi che il cinema abbia esattamente 5 sale e che non cambieranno mai nel tempo. Le sale presentano le stesse dimensioni cioè hanno tutte lo stesso numero di file e posti per ognuna di esse. La gestione delle sale è stata completamente rimossa.

Avvio applicazione



Il sistema viene avviato tramite un controller generale il quale si occupa di inizializzare la directory di lavoro del sistema , tramite `ManagerWorkingDir` e di gestire tutti i vari sotto controller , che espongono le varie funzionalità offerte dall'applicazione.

Il primo controller che deve essere avviato è quello che permette l'autenticazione dell'utente, quindi AccountController. Questo è l'unico che viene istanziato una e una sola volta, in quanto è necessario memorizzare l'account correntemente loggato.

Per quanto riguarda tutti gli altri controller, ogni volta che vi è la necessità di usufruirne vengono istanziati perchè non c'è il bisogno di tenerli in memoria tutti contemporaneamente.

CinemaController comunica con la view observable relativa al menù, della quale è observer.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per quanto riguarda la parte di testing si è deciso di utilizzare JUnit 5 in quanto studiato nel corso. I test sono stati scritti per controllare le parti più sensibili del codice ed esposte maggiormente a bug, le quali risultano fondamentali per il funzionamento di tutto il sistema. Questa fase ci ha permesso di avere una sicurezza aggiuntiva su quanto svolto.

3.1.1 Arianna Pagano

Testare la mia parte del progetto è stato molto importante in quanto costituisce una parte fondamentale per quanto riguarda tutta la gestione degli account. Per farlo ho deciso di sottoporre a test automatizzato il model degli account.

Le classi del test sono:

- Test Login che testa l'accesso corretto dell'username
- Test Manage Accounts che testa l'aggiunta di un nuovo account e l'eliminazione

3.1.2 Lorenzo Sansone

Una parte fondamentale della mia sezione è sicuramente il model dove vengono implementati meccanismi logici fondamentali per il funzionamento corretto del sistema.

- TestModelBooking: testa i principali metodi del BookingModel, più precisamente si testa i metodi per ritornare i posti prenotati di un film e le eliminazioni dei ticket quando i film o una sua programmazione sono eliminati
- TestSortDate: testa SorterByLocalDate che ordina in base alla data
- TestSortTime: testa SorterByLocalTime che ordina in base all'orario

3.1.3 Alessandro Zirondelli

Testare il mio codice all'interno del progetto è stato importante in quanto costituisce una parte fondamentale della logica di base del gestionale. Ho deciso quindi di sottoporre a test automatizzato il model dei films e dei films programmati. In particolare, per quanto riguarda la gestione dei films, ho effettuato diverse prove per verificare il corretto funzionamento dell'inserimento/rimozione dei films e della produzione/assegnazione degli ID per conto dell'Ids manager. Riguardo la programmazione invece, ho testato l'handler di gestione delle liste, per cui anche i relativi filtri e il model dei film programmati, soffermandomi in particolare sulla disponibilità della schedulazione.

Le classi di Test che ho scritto sono le seguenti:

-TestFilm

-TestFilmProgrammation

-TestHandlerList

3.2 Metodologia di lavoro

Per la realizzazione del sistema abbiamo deciso di utilizzare una metodologia di tipo Waterfall. Ci siamo prima occupati di un'analisi dei requisiti e del dominio applicativo. Dopodichè abbiamo proceduto con la progettazione, la quale ha occupato una buona parte di tempo. Una volta che le idee sono state chiarite siamo partiti individualmente con la scrittura del codice, dividendoci le varie funzionalità da implementare. In fase di progettazione ci è risultato fondamentale l'utilizzo degli UML per identificare i concetti base e per avere una visione generale di come sarebbe stato implementato il tutto. Come strumento di controllo di versione DVCS è stato utilizzato Git che ci è risultato utile nell'organizzazione del lavoro e nella revisione del codice. Abbiamo deciso di creare un branch per ogni funzionalità di cui ognuno è responsabile e tenerne uno solo, il master, sul quale fare affidamento per mantenere il codice corretto e funzionante.

Inizialmente era presente anche un quarto componente che poi per alcuni motivi non ha potuto più partecipare alla realizzazione di questo progetto. Data questa mancanza scoperta da parte nostra molto vicina alla data di consegna e per diversi problemi comunicativi, ci siamo dovuti riorganizzare per poter rimpiazzare una parte del suo lavoro. Ricoprire la totalità del suo lavoro sarebbe stata per noi impossibile gestirla in così poco tempo. Abbiamo quindi proceduto alla realizzazione di un Menù meno articolato senza gestire le operazioni sulle sale, avendole già inserite di default.

3.2.1 Arianna Pagano

Realizzazione del sistema di gestione degli account con relativo salvataggio, login. Realizzazione della parte di eliminazione e aggiunta degli account con i e relativi controlli generali. Implementazione delle statistiche.

3.2.2 Lorenzo Sansone

Realizzazione del model, controller e di tutte le view che riguardano la prenotazione dei biglietti. In particolare tutte le classi ed interfacce che appartengono ai package “booking” sono state implementate dal sottoscritto. Ho realizzato anche classi base come Ticket, TicketImpl, Row, Seat, SeatImpl, SeatState che si trovano nei package “utilities” e “utilitiesimpl”. Inoltre ho realizzato le classi per leggere e scrivere su file quindi tutte le classic he si trovano nei package “inputoutput”

3.2.3 Alessandro Zirondelli

Realizzazione completa, quindi comprensiva di view , model e controller relativa alla gestione dei film e alla programmazione. Quindi implementazione di tutte le varie funzionalità disponibili in questo ambito. Procedura di inizializzazione directory su cui l’applicazione si appoggia e preparazione dei file. Per precisazione ho svolto tutto il contenuto all’interno dei sottopackage denominati manageFilms, manageProgrammingFilms delle 3 componenti MVC. In aggiunta il contenuto del package exceptions , il sottopackage di utilities factory e ManaagerWotkingDir.

3.2.4 Alessandro Zirondelli e Lorenzo Sansone

Realizzazione del menu, del CinemaController e relative interfacce.

3.3 Note di sviluppo

3.3.1 Arianna Pagano

- Lambda expressions
- Stream
- Generici per la flessibilità del codice
- Optional soprattutto nella parte delle statistiche per evitare come valor di ritorno il null
- Libreria di Gson per il salvataggio degli utenti su file
- JUnit per i test
- Gradle, per supportare l'utilizzo di librerie esterne
- Programmazione funzionale

3.3.2 Lorenzo Sansone

- Stream utilizzati per rendere più leggibile e funzionale il codice.
- Optional utilizzati al posto di ritornare null
- Gson applicato per leggere e scrivere su file
- Uso di lambda expression
- Uso della libreria LocalDate e LocalTime per memorizzare date e orari
- Uso della libreria JPlanner per l'implementazione del calendario
- Utilizzo di Junit per i test
- Generici utilizzati nelle classi per scrivere e leggere su file
- Programmazione funzionale usata il più possibile soprattutto nel model
- Gradle per l'utilizzo di librerie esterne

3.3.3 Alessandro Zirondelli

- Programmazione funzionale, principalmente nella componente model all'interno dell'architettura
- Optional, per il salvataggio di alcune informazioni che sono appunto "Opzionali" e quindi non "strettamente rilevanti". La scelta è stata fatta per evitare possibili NullPointerException, una dei casi più comuni di eccezioni.
- Generici, sfruttati per il riuso e flessibilità del codice
- Gradle, per supportare l'utilizzo di librerie esterne e automatizzazione dello sviluppo
- JPlanner, libreria di MindFusion, utilizzata per alcune componenti grafiche
- Apache Commons, per la gestione directory e cartelle
- Modifiche apportate al file checkstyle.xml fornito in laboratorio, in quanto la sintassi non veniva interamente riconosciuta causa aggiornamento del plugin ad un'altra versione.

Capitolo 4

4.1 Autovalutazione

4.1.1 Arianna Pagano

Lavorare a questo progetto mi ha permesso di crescere e imparare molti aspetti della programmazione ad oggetti. Mi ha reso consapevole di difficoltà che si possono incontrare in progetti di dimensione maggiore a ciò che viene normalmente mostrato in aula. Ho scoperto quanto sia fondamentale una buona analisi iniziale e quanto non sia da sottovalutare ogni aspetto. Questo progetto mi ha permesso di rendermi conto quanta potenzialità abbia uno strumento come Git con cui non mi ero ancora interfacciata. Per il futuro infatti spero di poter imparare in maniera più approfondita le sue potenzialità. In conclusione sono in parte soddisfatta del mio lavoro in quanto nonostante mie mancanze, sono riuscita a concludere la mia parte.

4.1.2 Lorenzo Sansone

Pur avendo completato tutta la mia parte non sono molto soddisfatto del risultato anche a causa della mia scarsa esperienza in materia infatti dato che questo è il primo progetto molte cose le ho dovute cercare per conto mio togliendo tempo allo sviluppo. Ho cercato comunque di adottare tutte le “best-practise” possibili per quanto riguarda la scrittura e manutenzione del codice. Una parte importante è stata la grafica che ha portato via non poche ore per renderla più user-friendly possibile. Ho imparato sicuramente quanto sia importante lavorare in gruppo e sostenersi a vicenda per progredire con il progetto. Per quanto riguarda l’uso di Git penso di aver appreso i comandi base per poter lavorare a progetti. Credo di essere stato attivo durante la parte iniziale della progettazione dell’applicazione.

4.1.3 Alessandro Zirondelli

Complessivamente sono contento del lavoro che siamo riusciti a fare. Personalmente non sono completamente soddisfatto del mio operato e di alcuni aspetti di organizzazione del team. Questo progetto mi ha aiutato a capire l’enorme importanza della fase di progettazione e del processo di lavoro/ organizzazione che è necessario fare per una buona realizzazione di un lavoro , anche nell’ottica di futuri interventi. Ho cercato infatti di dedicare, assieme ai miei compagni, una buona parte di tempo per la progettazione. Purtroppo mi sono reso conto però, durante l’avanzamento del progetto, che alcune cose da parte mia, potevano essere progettate e pensate meglio fin dall’inizio. Questo mi ha portato quindi a fare un passo indietro per cercare di rimediare alla mancanza. Nonostante questo però sono riuscito, a mio parere , a fare un discreto lavoro.

4.2 Commenti finali

4.2.2 Lorenzo Sansone

Il progetto non è stato per niente leggero da affrontare questo grazie anche al fatto che non avendo esperienza all'inizio facevo fatica anche ad immaginare come partire a costruirlo.

Nonostante questo progetto mi abbia dato molto alla fine non nascondo che il prezzo da pagare per raggiungere l'obiettivo è stato eccessivo infatti troppo spesso mi sono ritrovato a spendere molto tempo per studiare in maniera approfondita una serie di argomenti che erano stati spiegate in poche ore di lezione e che magari meritavano più tempo. Il tutto è reso più difficile dovendo seguire i corsi delle altre materie. Secondo me la situazione potrebbe migliorare se si illustrasse un esempio vero di progetto durante le lezioni e se si incrementasse la documentazione per quanto riguarda UML e MVC.

4.2.3 Alessandro Zirondelli

Data la mia pochissima esperienza è stato davvero complicato questo corso, in particolare il progetto. Credo che il quantitativo di ore di studio necessarie sia veramente troppo alto a mio parere, considerando che sono presenti anche altre materie di studio e altri esami. Sicuramente penso che sia un ottimo corso, ma credo che la preparazione richiesta che si punta a raggiungere sia troppo alta in così breve tempo. Nonostante questo sono soddisfatto di questo corso perchè mi ha insegnato diversi altri aspetti, oltre a quelli inerenti alla programmazione ad oggetti, che mi serviranno in futuro.

Appendice A

Guida Utenti

A.1 Avvio dell'applicazione

All'avvio dell'applicazione, il sistema richiede l'autenticazione. Se si tratta del primo accesso, le credenziali di amministrazione sono le seguenti:

Username: admin

Password: 123

Account operatore default:

Username: operator

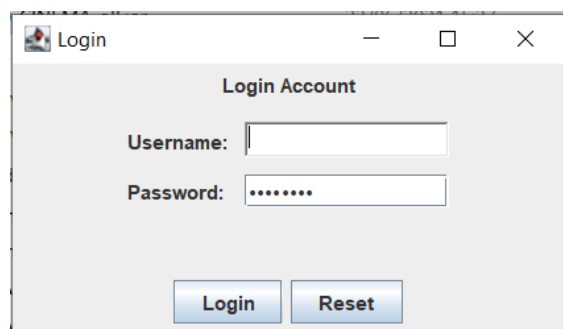
Password: 123

Per usufruire della modalità “demo”, nella quale sono stati inseriti diversi dati è necessario loggarsi con le seguenti credenziali:

ATTENZIONE: Se si continua ad utilizzare l'account demo anche nei successivi login i dati rimarranno sempre I medesimi per cui ogni modifica apportata non verrà salvata, in quanto si ripristinerà la situazione dei dati fornita per la demo. Se si usa l'account demo dopo aver modificati i dati nel programma autotenticandosi con un altro utente questi verranno persi.

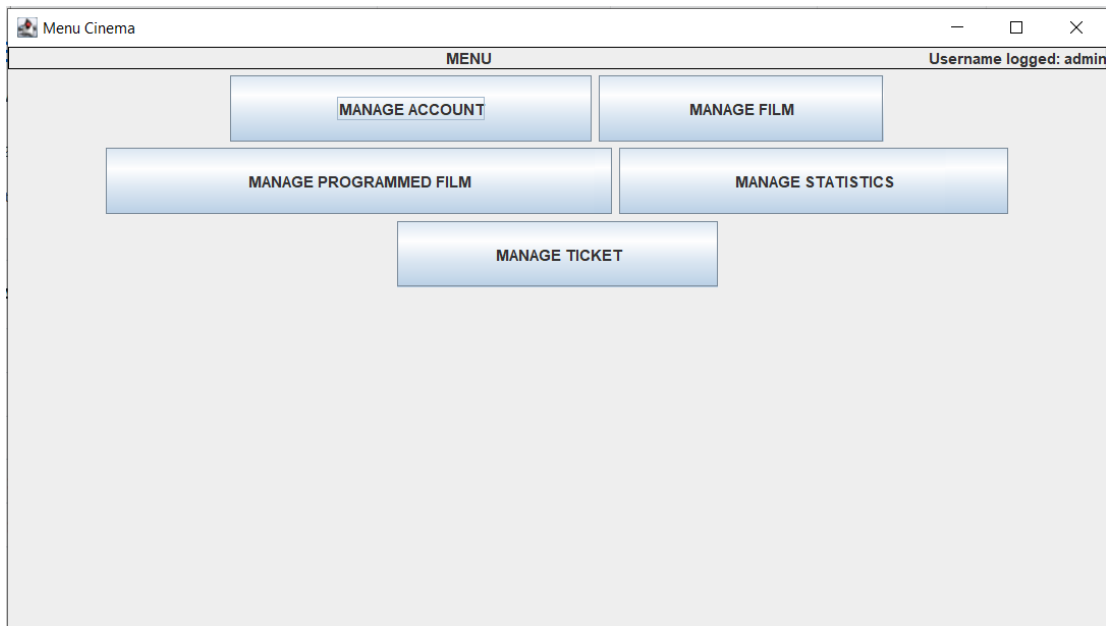
Username: demo

Password: 123



A.2 Menù principale

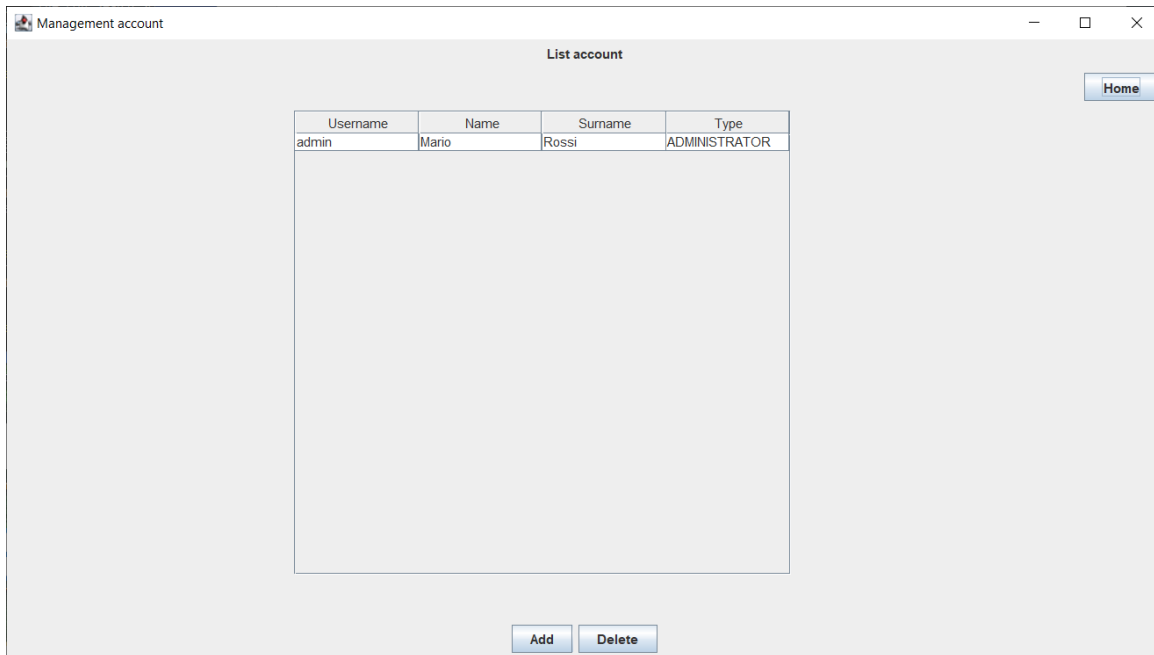
In base all'autenticazione, se operatore o amministratore, il menù abilita o disabilita le varie funzioni disponibili mediante i bottoni. Possiamo notare che l'unica operazione disponibile se si è operatori è la gestione dei ticket. Tutte le altre operazioni, ovvero la programmazione dei film, l'inserimento dei film, la gestione degli account e le statistiche, sono gestibili solo dall'amministratore. Cliccando su un bottone, si aprirà la relativa schermata di gestione. In alto a destra è possibile visualizzare l'username dell'account loggato.



A.3 Fuzionalità

A.3.1 Gestione Account

Qui vengono mostrati gli account registrati nel sistema attraverso una tabella esplicativa che mostra l'username, il nome, il cognome e la tipologia dell'account. In basso sono presenti due bottoni che permettono di aggiungere o rimuovere un account esistente.



A.3.1.1 ADD

ADD visualizza una nuova schermata di inserimento dei dati dell'account. Qui devono essere inserite diverse informazioni che risultano obbligatorie per la corretta registrazione. Tra cui di particolare importanza la selezione della tipologia dell'account.

The screenshot shows a window titled "Registration" with a subtitle "Add account". It contains the following form fields:

- Username:
- Name:
- Surname:
- Password:
- Repeat Password:
- Type: (dropdown menu)

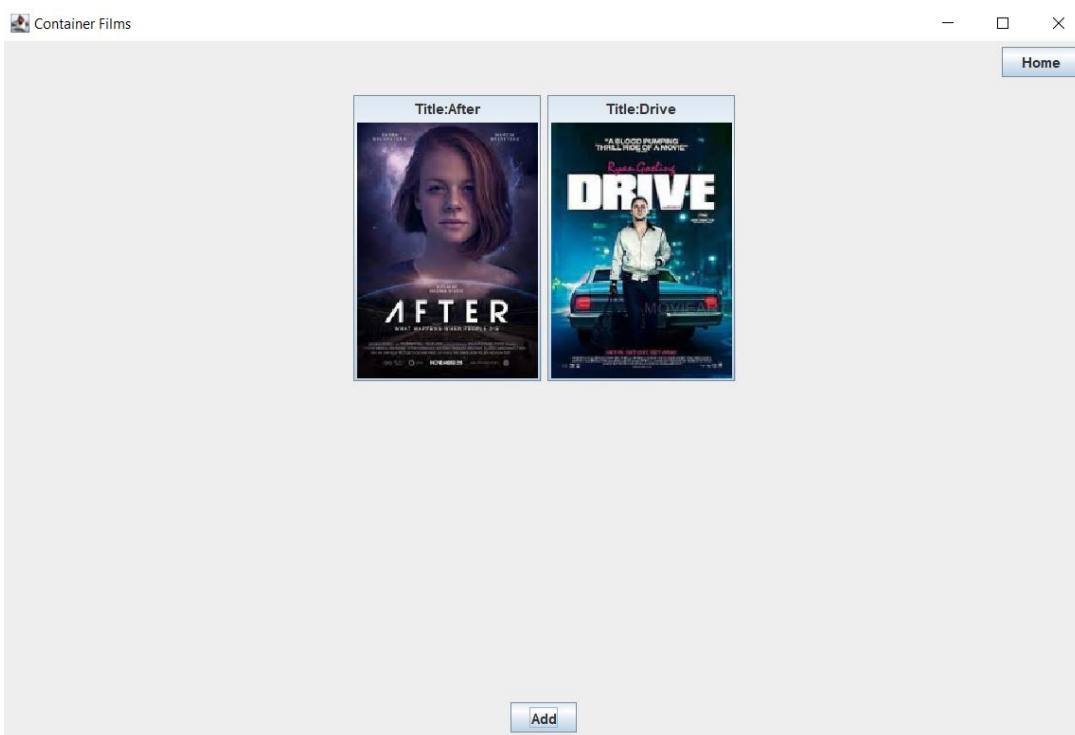
At the bottom are three buttons: "Save", "Reset", and "Cancel".

A.3.1.2 DELETE

DELETE permette l'eliminazione di un solo account per volta dopo aver selezionato la riga corrispondente nella tabella

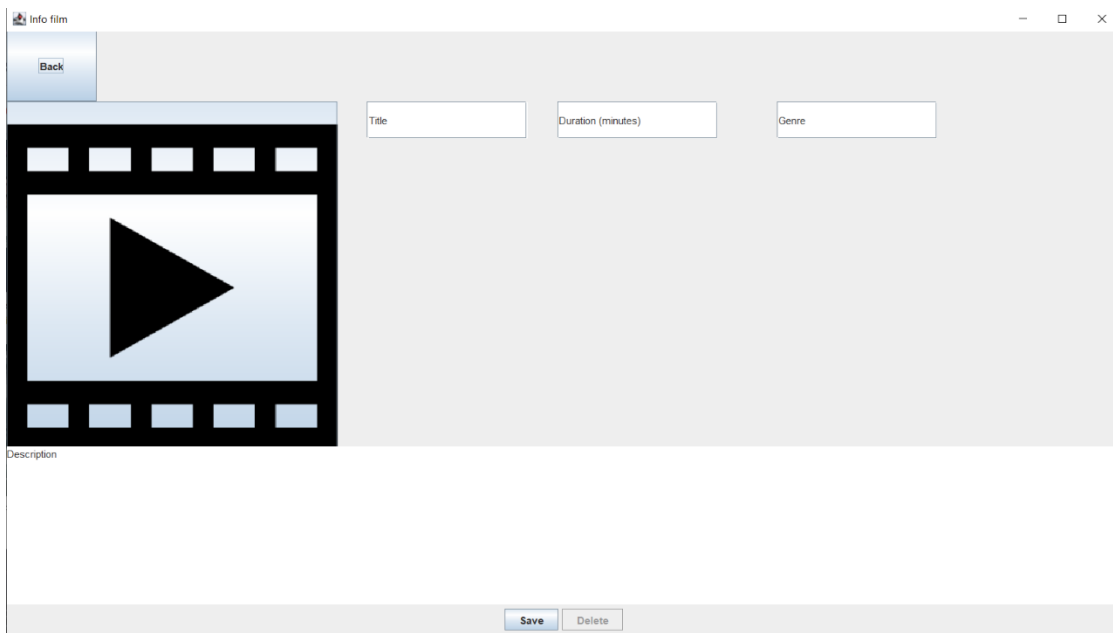
A.3.2 Gestione film

Qui è possibile visualizzare tutti i film che il cinema ha voluto inserire e che quindi sono programmabili. Se durante l'apertura di questa schermata non è presente alcun film, allora verrà mostrata una finestra di pop-up con quanto appena detto. In basso è presente il bottone "Add" che permette di procedere con l'inserimento di un nuovo film. Cliccando su un qualsiasi film, se presenti, vengono mostrate le sue informazioni.



A.3.2.1 ADD

Il pulsante Add visualizza una schermata in cui è possibile aggiungere le informazioni relative al film. Opzionalmente è possibile aggiungere anche una immagine di copertina che è possibile importare e caricare in locale. In basso sono presenti i bottoni “Save” e “Delete”. Cliccando sul bottone relativo all’immagine di copertina sarà possibile tramite un’apposito schermata selezionare l’immagine desiderata ed importarla cliccando sul bottone Open,



The screenshot shows a window titled "Info film" with a standard Windows-style title bar (minimize, maximize, close buttons). Inside the window, there is a "Back" button in the top-left corner. Below it is a large rectangular area with a film strip border and a large black play button in the center, intended for a movie poster. To the right of this area are three text input fields labeled "Title", "Duration (minutes)", and "Genre". Below the poster area is a large text area labeled "Description". At the bottom of the window, there are two buttons: "Save" and "Delete".

A.3.2.1.1 SAVE


SAVE permette di salvare le informazioni e procedere con l’inserimento.

A.3.2.1.2 DELETE

DELETE permette di eliminare un film che è già stato inserito nel sistema. Questo bottone è disabilitato se si sta procedendo con l'inserimento di un nuovo film. E' abilitato invece se si è in fase di consultazione delle informazioni per cui è già stato inserito in passato.

Info film

Back



Belissimo

After

60

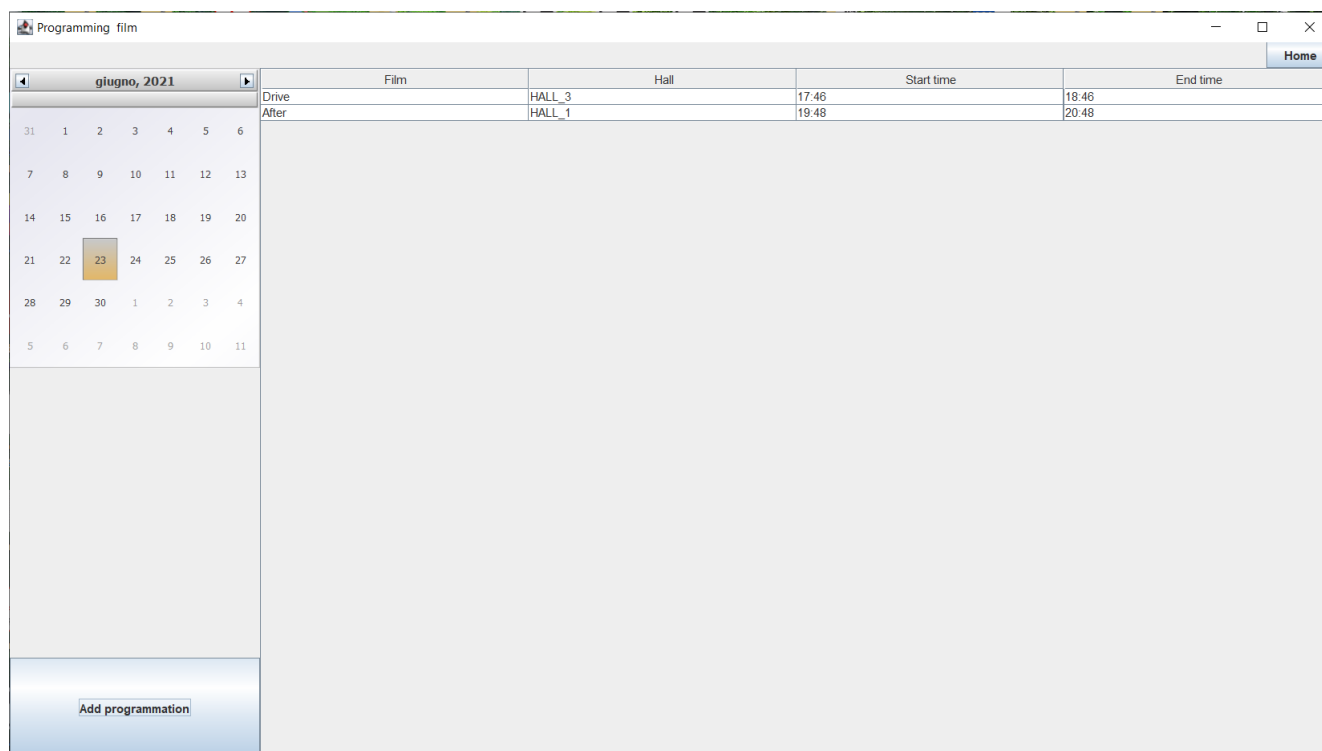
Fantasy

Save

Delete

A.3.3 Gestione film programmati

Questa schermata mostra sulla sinistra un calendario tramite il quale è possibile navigare nelle programmazioni relative alla data selezionata e un bottone per aggiungere una programmazione per un film già registrato precedentemente. Sulla destra si può vedere una tabella riassuntiva con le programmazioni relative al giorno selezionato sul calendario.



Film	Hall	Start time	End time
Drive	HALL_3	17:46	18:46
After	HALL_1	19:48	20:48

A.3.3.1 ADD

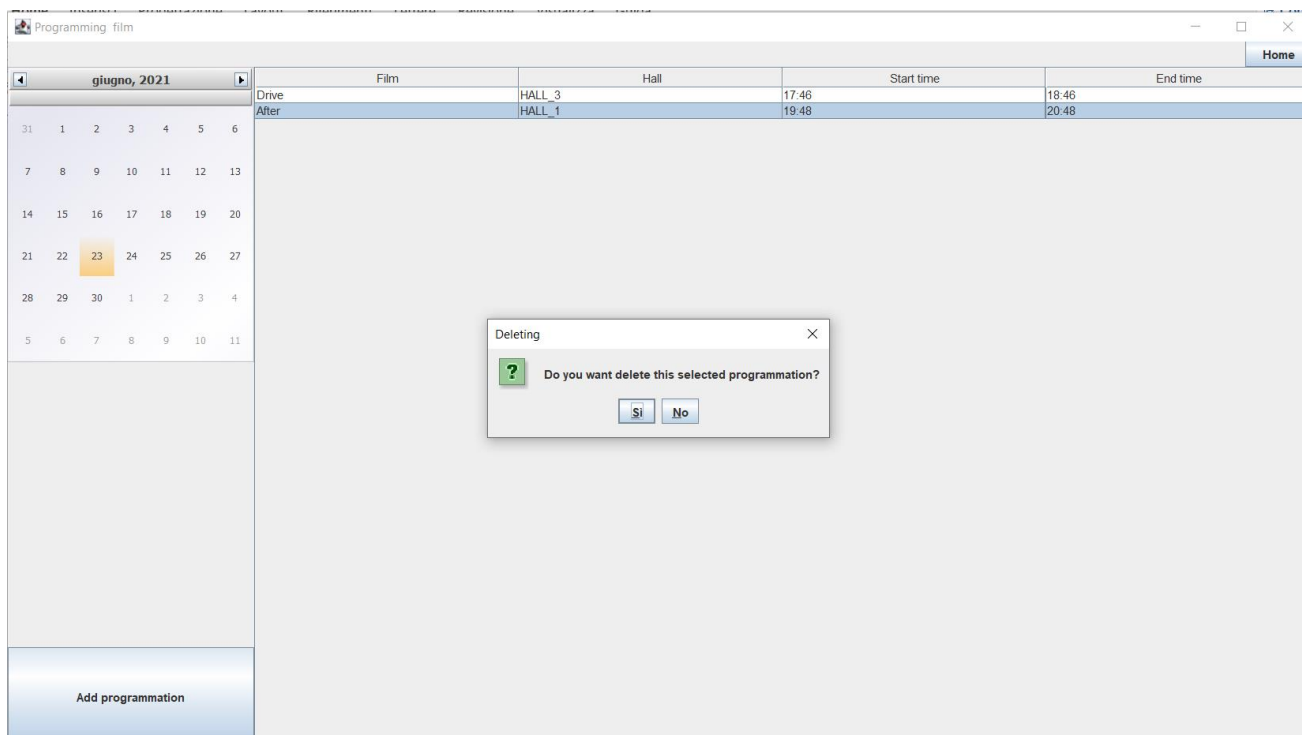
In questa schermata è possibile selezionare i dati relativi alla programmazione del film, quindi: data, orario di inizio, sala e film da schedare. Non è necessario inserire l'orario di fine in quanto viene automaticamente calcolato in base alla durata del film. Per procedere con la programmazione è necessario cliccare il tasto Schedule.

The screenshot displays the 'Programming film' application window. On the left, there is a calendar for June 2021. The date 23 is highlighted. Below the calendar is a button labeled 'Add programming'. The main area of the window is a table with columns: Film, Hall, Start time, and End time. A 'Home' button is in the top right corner. A 'Schedule a film' dialog box is open in the center. It contains the following fields:

- Date and start time**
 - Enter Date**: Month (giugno), Day (23), Year (2021)
 - Enter Time**: Hour (16), MIN (46)
- Info**
 - Price: 3.4 e.g. (euro)
 - Hall: (dropdown menu)
 - Film: (dropdown menu)
- Schedule** button

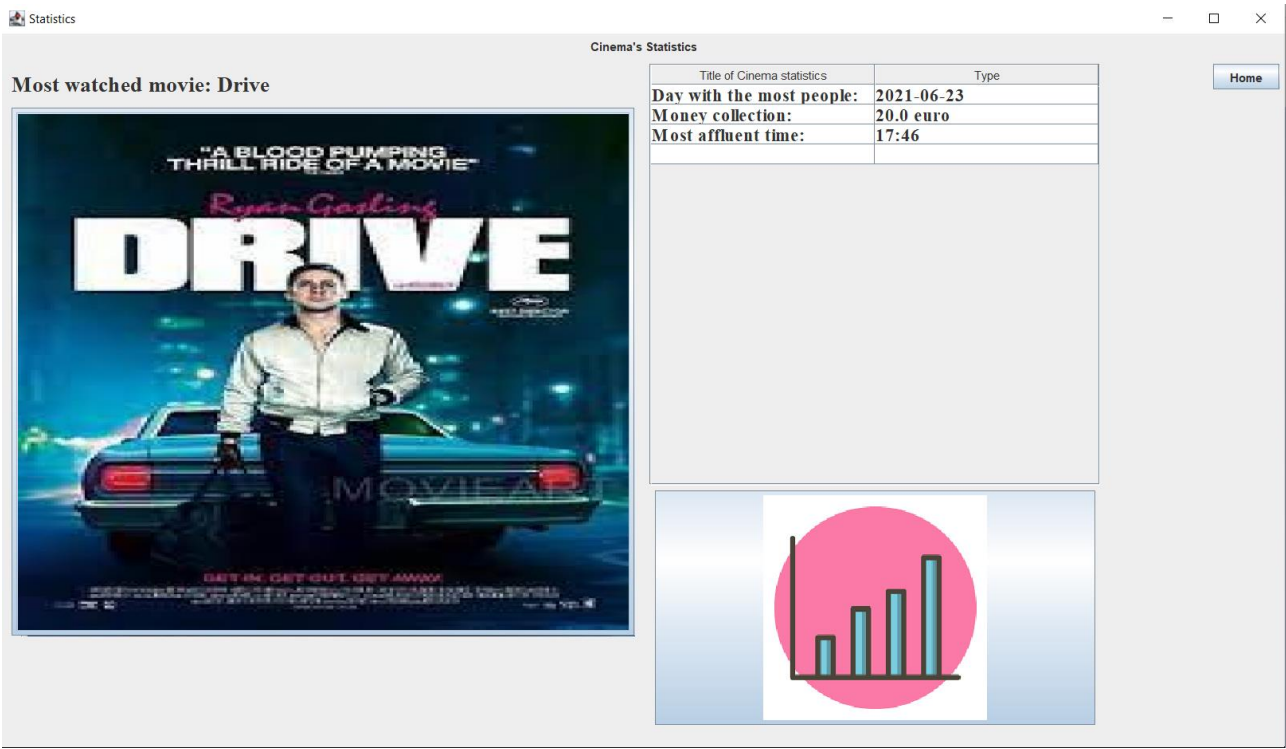
A.3.3.2 Delete

Se sono presenti già delle programmazioni, facendo doppio click sulla riga corrispondente, apparirà un pop-up con la possibilità di eliminarla.



A.3.4 Statistiche

Qui è possibile visualizzare tutte le statistiche base relative alle vendite del cinema. A sinistra è presente il titolo del film più visto e subito sotto la sua copertina, se questa è presente. A destra invece è presente una tabella in cui vengono mostrate altre tre statistiche con nella prima colonna la descrizione e affianco il risultato

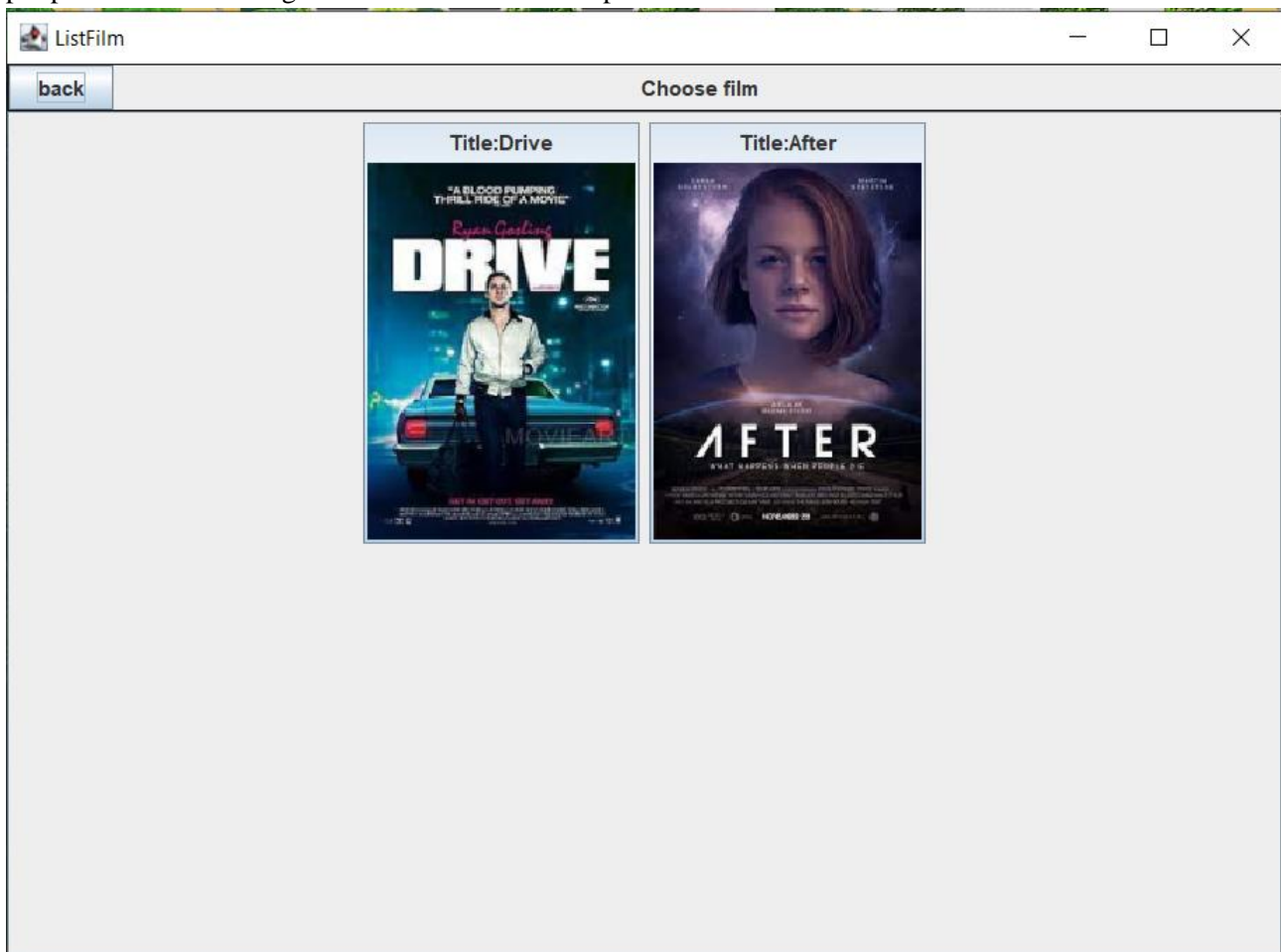


A.3.5 Gestione prenotazioni posti a sedere

Qui è possibile prenotare i posti per un film in programma e ovviamente vedere quali sono già stati comprati in passato. Questa sezione è divisa in tre fasi: scelta del film, scelta della programmazione e scelta dei posti a sedere per quella programmazione

A.3.5.1 Scelta dei film

Da questa schermata si possono scegliere i film per cui è possibile continuare con la prenotazione. Quindi per procedere oltre bisogna cliccare su una delle copertine esistenti altrimenti l'interfaccia risulterà vuota



A.3.5.2 Scelta della programmazione

Dopo aver scelto il film si seleziona una sua programmazione dalla tabella esposta. All'inizio sono mostrate tutte le programmazioni di quel film ma se è necessario si possono ordinare e/o filtrare grazie al pannello presente nella parte sinistra dell'interfaccia.

Select a schedule from: Drive		
Date	Time	Hall
2021-06-23	17.46	HALL_3

A.3.5.2.1 Tasto Apply

Selezionando una data e/o spuntando un sorter(data o time) e cliccando il tasto apply si filtrano i film programmati disponibili secondo le specifiche selezionate.

A.3.5.2.2 Tasto Reset

Mostra semplicemente tutte le programmazioni per il film scelto quindi se erano stati applicati dei filtri viene ripristinata la situazione iniziale

A.3.5.2.3 Tasto Select

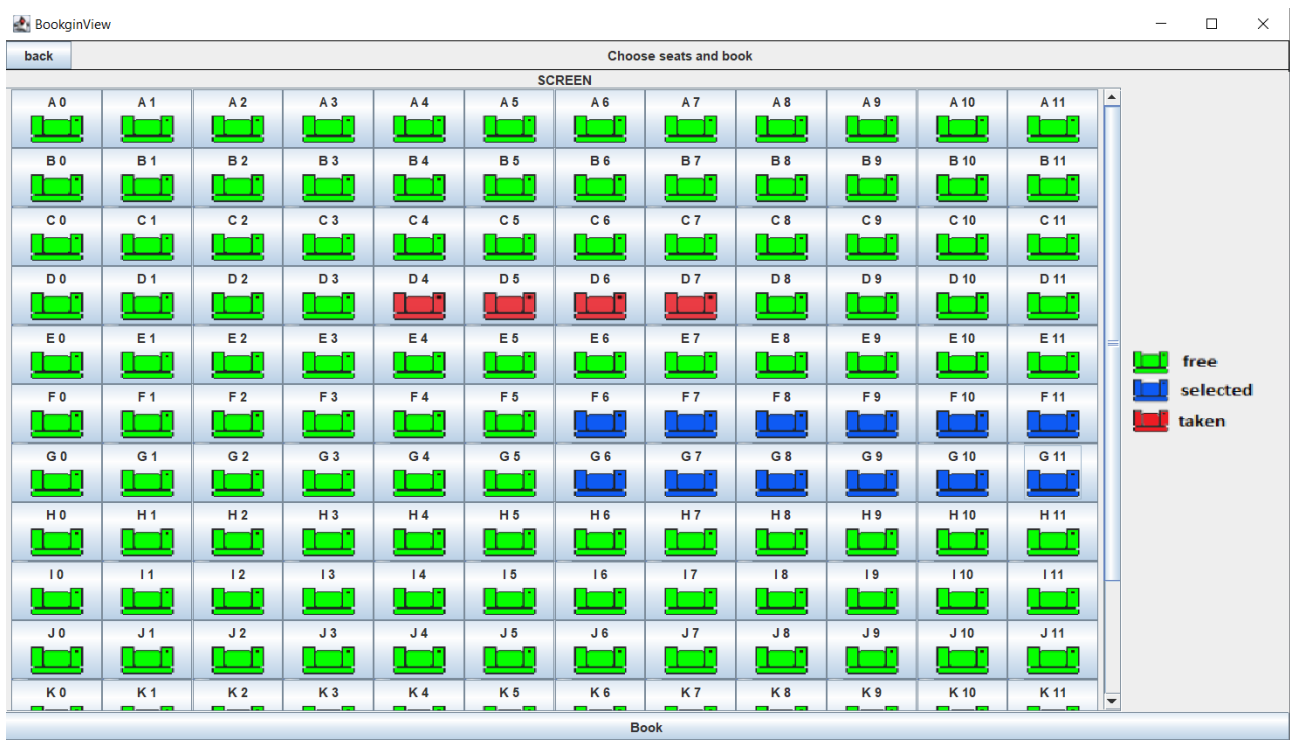
Dopo aver selezionato un film in programma, quindi una riga di una tabella, si può procedere con il tasto select che provvederà a mostrare la schermata successiva

A.3.5.2.4 Tast Back

Si ritorna alla schermata precedente

A.3.5.3 Scelta dei posti a sedere

In questa schermata si potrà iniziare la prenotazione dei posti selezionando quelli di interesse. I posti colorati di verde sono quelli disponibili, quelli blu sono quelli correntemente selezionati e quelli rossi sono quelli prenotati



A.3.5.3.1 Tasto Book

Una volta selezionati i posti premendo il tasto book si prenoteranno e risulteranno colorati di rosso

A.3.5.3.2 Tasto Back

Si ritorna alla schermata precedente