Theatre Manager

Programmazione ad Oggetti
AA. 2014-2015

Simone Romboli
28/02/2015

Indice

1. Analisi

1.1. Requisiti

L'applicazione che si intende sviluppare vuole simulare la gestione di una biglietteria di una sala per eventi (cinema, teatro, conferenze, …). Dovrà perciò permettere sostanzialmente di gestire la configurazione della sala, prenotare e rimuovere biglietti, informare qualora la sala sia piena, e così via.

1.2. Problema

L'applicazione dovrà gestire un evento e la sala (o le sale) nella quale esso si svolge interfacciandosi con l'utente utilizzatore in una maniera chiara e diretta.
Dovrà essere possibile personalizzare la configurazione della sala (in termini di file e posti a sedere), caratteristiche dell'evento (a ingresso libero o a pagamento, con o senza prenotazione del posto).
Potrà essere utile salvare le informazioni relative a un evento su memoria per poi riprenderne la gestione in un secondo momento.

2. Design

2.1. Architettura

A livello di design architetturale dell'applicativo si è cercato di adottare il pattern Model-View-Controller. All'utente utilizzatore è mostrata un'interfaccia grafica (View) la quale invia dati e richiede informazioni comunicando con il controller, incaricato di reperire quanto richiesto dal modello.

2.2. Design dettagliato

Nella figura seguente sono mostrati i principali concetti con i quali l'applicazione ha a che fare.
Sono stati separati ciò che modella un evento e le relative informazioni da l'idea di sala nella quale si svolge e tutta la gestione dei posti ad essa correlata. Sono date le implementazioni delle interfacce "IEvent" e "Theatre" che descrivono le funzionalità richieste dai relativi concetti nelle classi "Evento" e "Platea". Il modello principale che l'applicazione gestisce è dunque un "RealModel" ossia la sintesi delle due unità base: si compone di un campo per le informazioni legate all'evento e di un insieme di sale ad esso collegate.

Non si evidenziano particolari pattern di progettazione applicati. C'è stata la volontà di provare ad applicare alcuni dei pattern noti, ma il mancato polso e precisione in fase implementativa ha fatto in modo che ci siano i presupposti per evidenziare meglio i pattern in una futura fase di miglioramento ma che, realmente, essi non siano presenti.

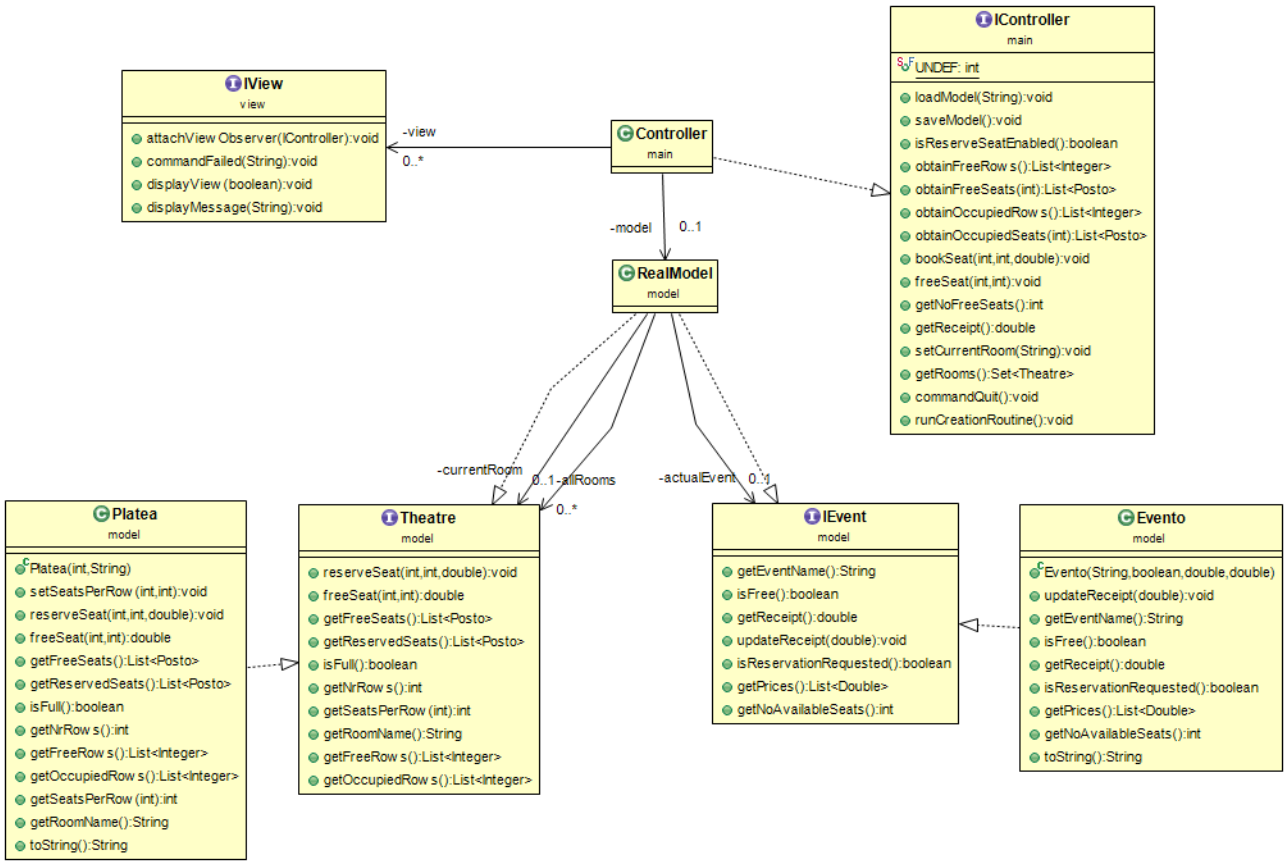
IView
view
attachView Observer(IController):void
commandFailed(String):void
displayView (boolean):void
displayMessage(String):void
-view
0..*
Controller
main
IController
main
UNDEF: int
loadModel(String):void
saveModel():void
isReserveSeatEnabled():boolean
obtainFreeRows():List<Integer>
obtainFreeSeats(int):List<Posto>
obtainOccupiedRows():List<Integer>
obtainOccupiedSeats(int):List<Posto>
bookSeat(int,int,double):void
freeSeat(int,int):void
getNoFreeSeats():int
getReceipt():double
setCurrentRoom(String):void
getRooms():Set<Theatre>
commandQuit():void
runCreationRoutine():void
-model
0..1
RealModel
model
-currentRoom
0..1
-allRooms
0..*
-actualEvent
0..1
Platea
model
Platea(int,String)
setSeatsPerRow (int,int):void
reserveSeat(int,int,double):void
freeSeat(int,int):double
getFreeSeats():List<Posto>
getReservedSeats():List<Posto>
isFull():boolean
getNrRows():int
getFreeRows():List<Integer>
getOccupiedRows():List<Integer>
getSeatsPerRow (int):int
getRoomName():String
toString():String
Theatre
model
reserveSeat(int,int,double):void
freeSeat(int,int):double
getFreeSeats():List<Posto>
getReservedSeats():List<Posto>
isFull():boolean
getNrRows():int
getSeatsPerRow (int):int
getRoomName():String
getFreeRows():List<Integer>
getOccupiedRows():List<Integer>
IEvent
model
getEventName():String
isFree():boolean
getReceipt():double
updateReceipt(double):void
isReservationRequested():boolean
getPrices():List<Double>
getNoAvailableSeats():int
Evento
model
Evento(String,boolean,double,double)
updateReceipt(double):void
getEventName():String
isFree():boolean
getReceipt():double
isReservationRequested():boolean
getPrices():List<Double>
getNoAvailableSeats():int
toString():String

fig. 1

3. Sviluppo

3.1. Testing automatizzato

Su alcune parti di modello di è provveduto a testing automatizzato tramite la suite JUnit.
Nello specifico alcuni test sono stati eseguiti per verificare la correttezza delle informazioni relative alla sala, ai posti disponibili e all'incasso a fronte di ripetute prenotazioni o rimozioni di posti.

3.2. Divisione dei compiti

Essendo stato svolto il progetto in singolo non c'è stata alcuna divisione dei compiti.
È stato impiegato il DVCS Mercurial per il versioning dell'applicazione insieme ad un repository Bitbucket.

3.3. Note di sviluppo

Per la realizzazione della classe "CreationDialog.java" è stato utilizzato il plug-in "Window Builder" per Eclipse. Per questo motivo la classe richiede l'impiego di una libreria esterna "forms 1.3.0" per la creazione del layout della GUI che essa crea.
Non c'è una motivazione particolare sul perché si è voluto usare Window Builder dato che la GUI che crea la classe non è complessivamente articolata. Si è semplicemente voluto sperimentare l'utilizzo del suddetto plug-in.
L'applicazione è stata testata con due differenti sistemi operativi: Windows 8.1, Linux Mint 17.1.

3.4 Bug Noti

1. Per ragioni di tempo non è correttamente implementata la funzione di rimozione di un posto prenotato nel caso non sia prevista la prenotazione del posto ma venga assegnato automaticamente.

2. In fase di salvataggio le informazioni relative ai prezzi vengono perse siccome per le categorie dei prezzi è stata impiegata una Enum che però in fase di serializzazione perde ogni informazione personalizzata associata[1]. Tutto ciò è stato scoperto ormai troppo tardi per permettere un adeguato refactory, impiegando semplici campi double e relative chiamate su di essi. Ne consegue che caricando un evento salvato e provando a prenotare non avviene nessuna variazione nell'incasso.

4. Commenti finali

4.1. Conclusioni

Questo progetto è stato effettuato esclusivamente per l'insegnamento di Programmazione ad Oggetti. Non c'è al momento alcuna intenzione di migliorarlo ed estenderlo.
Come si evince facilmente da una esecuzione dell'applicazione questa risulta abbastanza banale e poco articolata sebbene con i dovuti accorgimenti le funzionalità minime richieste le soddisfa.

4.2. Difficoltà incontrate e commenti per i docenti

Sebbene alcune ore siano state dedicate alla modellazione del applicazione in alcuni casi sono stato costretto a modificare totalmente gli oggetti con i quali lavorare, probabilmente per una visione troppo elementare dei problemi da affrontare e della relativa soluzione totalmente in contrasto con le problematiche implementative di un linguaggio di programmazione (o magari viceversa non sono stato abbastanza bravo ad analizzare il problema e ad implementare quanto pensato).
Un fattore determinante nella riuscita parziale del progetto è stato scegliere la prima deadline disponibile a fronte di un limitato tempo da dedicare al progetto per via di impegni personali extra universitari presenti lungo il mese di febbraio. In tal modo non sempre l'attenzione era focalizzata sul progetto e forse anche per questo si sono verificati i problemi cui sopra. Potrei anche affermare che forse qualora avessi scelto la deadline successiva probabilmente il risultato non sarebbe stato migliore dato che mi sarei dovuto preoccupare anche di altri insegnamenti (e altri impegni).
Lavorare in singolo a seconda degli ambiti ha i suoi pregi e i suoi difetti: in questo caso ha sicuramente limitato la buona riuscita del progetto

impedendone una buona progettazione ed implementazione, un confronto avrebbe molto probabilmente migliorato il risultato finale.