Progetto di Ingegneria del Software  
 Parte 2   
a.a. 2017/2018

**REFACTORING ARCHIVIO MULTIMEDIALE**

*RELAZIONE ATTIVITA’ SVOLTE*

Landi Federico

Prandini Stefano

# 1) PRINCIPIO DI SEPARAZIONE MODELLO-VISTA

Il primo passo del nostro lavoro di refactoring del codice è stato quello di separare la logica di business della nostra applicazione dalla logica di presentazione.

Per raggiungere questo obiettivo abbiamo pensato di applicare il pattern architetturale MVC (Model-View-Controller):

* *Model* (logica di business): rappresenta tutte le classi che hanno a che fare con il campo d'utilizzo dell'applicazione e non con l'interazione con l’utente.
* *View* (logica di presentazione): è l’interfaccia grafica dell’applicazione. Nel nostro caso, essendo un’applicazione da riga di comando, si riduce alle istruzioni di stampa a video.
* *Controller* (logica di controllo): è la parte che fa da interfaccia tra il *Model* e la *View*. Rappresenta un oggetto artificioso che coordina le operazioni di sistema dell’applicazione.

La cosa fondamentale è che il *Model* non dipenda dalla *View*, in modo che, se in una futura implementazione si decidesse di introdurre una GUI, la logica di business potrà venire riutilizzata in tutto e per tutto.

Per questo motivo abbiamo deciso di tenere nel *Model* tutte le strutture dati e i metodi che operano su di esse, a patto che non richiedano un’interazione con l’utente o con la parte grafica. I metodi che erano nel *Model* e che richiedevano una di queste due componenti sono stati spostati nei *Controller*.

I metodi che stampavano le informazioni di un determinato oggetto (risorsa, fruitore, prestito, …)sono invece stati modificati in metodi *toString(),* che verranno stampati dalle rispettive *View*.

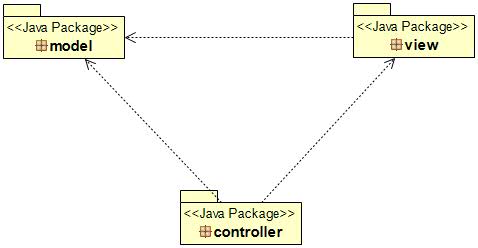
Abbiamo creato diverse classi *Controller* e diverse classi *View*, in modo da avere un triangolo *Model*-*View*-*Controller* per ogni “oggetto di dominio”, in particolare:

* Libro/Libri -> LibriView -> LibriController
* Film/Films -> FilmsView -> FilmsController
* Fruitore/Fruitori -> FruitoriView -> FruitoriController
* Prestito/Prestiti -> PrestitiView -> PrestitiController
* Storico -> StoricoView -> StoricoController
* Archivio -> ArchivioController
* RisorseView, per unire metodi di FilmsView e LibriView uguali (stampa “libro” o “film” a seconda del caso)

Ogni oggetto *Controller* possiede un attributo corrispondente al proprio *Model*, in modo da poter operare sui dati veri e propri dopo aver interagito con l’utente.  
Il *Controller* (collaborando con la *View*) interagisce con l’utente e modifica/aggiorna i dati del *Model*.

La mancanza della dipendenza da *Model* a *View* è confermata, a livello di codice, dal fatto che nessuna classe del package *Model* importa alcuna classe del package *View*.

Ciò si vede anche dal diagramma UML dei package, nel quale la *View* dipende solo dal *Model*, il *Controller* dipende sia da *Model* che da *View* (dovendo interagire con entrambi) e il Model è indipendente da tutto il resto.



# 2) OPERAZIONI DI SISTEMA (responsabilità e pattern GRASP)

Caso d’uso “Registrazione”:

caso d’uso

|  |  |
| --- | --- |
| NOME | **REGISTRAZIONE** |
| ATTORE | FRUITORE |
| SCENARIO  PRINCIPALE | 1. L’utente sceglie la funzionalità “Registrazione” 2. Il sistema presenta i campi anagrafici da compilare 3. L’utente inserisce i dati anagrafici 4. Il sistema presenta il campo Username 5. L’utente inserisce il dato 6. Il sistema presenta due volte il campo Password 7. L’utente completa i due campi 8. Il sistema stampa “Registrazione avvenuta con successo”   FINE |
| SCENARIO  ALTERNATIVO | 4a. (Precondizione: la data inserita non esiste o è futura)  Il sistema mostra un messaggio di errore e chiede di reinserire la data  TORNA AL PUNTO 2 |
| SCENARIO  ALTERNATIVO | 4a. (Precondizione: l’utente non è maggiorenne)  Il sistema informa l’utente che non può registrarsi (e termina la procedura di registrazione)  FINE |
| SCENARIO  ALTERNATIVO | 6a. (Precondizione: username non disponibile)  Il sistema indica all’utente di scegliere un altro username  TORNA AL PUNTO 4 |
| SCENARIO  ALTERNATIVO | 4a. (Precondizione: le 2 password non coincidono)  Il sistema informa l’utente che le due password inserite non coincidono  TORNA AL PUNTO 6 |

# SSD:

SISTEMA

UTENTE

AddFruitore

msgChiediNome

Enter nomee

msgChiediCognome

WHILE

Enter Cognome

! DataValida

Enter data

msgChiediDataNascita

IF

WHILE

msgChiediUsername

Utente Maggiorenne

! User

Disponibile

Enter Username

IsUsernameDisponibile

WHILE

msgChiediPSW1

! pswUguali

Enter PSW1

msgChiediPSW2

Enter PSW2

pswUguali

msgConfermaDati

Conferma dati

# VIEW

Si potrà creare una specie di controller grasp per creare e gestire tutti i controller?  
MainController, credo rappresenti qualche pattern.

Si potrebbe fare un Controller Grasp (che funziona anche da FACADE: facade controller) che riceva tutti gli eventi di sistema dalla UI (ossia dal nostro main?) e che poi deleghi i compiti ai singoli controller MVC. Servirebbe per il punto 2). (vedi CONTROLLER GRASP)  
*“Facades are meant to be wrappers around complex functionality, their primary goal is hiding complexity of an underlying system. You can think of the Facade as a layer wrapping the complex functionality and providing simpler methods to interact with.   
A controller, on the other hand, is normally tied to a very specific piece of functionality in the system. Its main goal is to mediate the interaction between a view and a model, or in some cases just a model.  
It's more typical to have a facade interacting with several controllers, than vice versa.”*

ArchivioController fa qualcosa del genere con Film e Libri.

Le nostre view sono statiche, non ha senso istanziare classi che stampino solo su console (?)  
Non vanno quindi passate come parametro ai controller. Oppure le istanzio lo stesso (come singleton probabilmente) in modo che se ci sarà un’interfaccia utente basterà sostituirla? Dovrà implementare un’interfaccia che comprenda tutti i metodi, in modo che non ci sarebbero problemi nella sostituzione?



# MENU

Creata una classe per ogni menu, tolte tutte le parti di grafica e spostate nel package *view*.

Sono tutti Extract Class (del refactoring) ?

Anche per la grafica sono tutti extract method: i syso diventano metodi nella view (quindi anche move method)

Sono classi statiche, cioè non vengono istanziate ma hanno metodi statici.

Potrebbero diventare singleton? Così si possono istanziare ma solo una volta.

# EVENTI/INTERAZIONI SISTEMA (da casi d’uso) per SSD

* Verifica operatore
* Visualizza storico
* Visualizza fruitori
* Visualizza prestiti

Operatore

* Aggiungi risorsa (libro/film)
* Rimuovi risorsa (libro/film)
* Visualizza risorse(libri/film)
* Cerca una risorsa(libro/film)
* Login
* Registrazione
* Visualizza dati personali
* Rinnova iscrizione

Fruitore

* Termina prestiti
* Richiedi un prestito
* Rinnova prestito
* Visualizza prestiti in corso
* Cerca una risorsa(libro/film)

# REFACTOR

IN MenuRichiediPrestito c’era codice duplicato per le varie risorse: usando la superclasse Risorsa sono riuscito a ridurlo molto, lasciandolo solo per distinguere tra il numero massimo di risorse prestabile tra libri e film (si dovrebbe sistemare pure questo, un po’ dappertutto)

Storico da classe statica è diventato istanziabile: al costruttore si passano archivio ecc. e quando serve si può interrogare per avere informazioni storiche.  
In tutti i metodi di StoricoController viene utilizzato il relativo model (Storico). La differenza sta che StoricoController interagisce con utente.

Tolti metodi statici get e set per utenteLoggato nel main, sennò era una variabile globale. Ora viene passata l’istanza dell’utente loggato tra i metodi.

# PATTERN

Singleton con parametri: <https://paolopatierno.wordpress.com/2011/07/06/pattern-singleton-con-parametri/>