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:

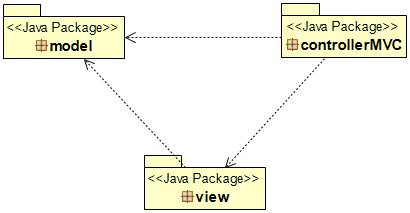
* Risorse -> RisorseView -> RisorseController
* Libro -> LibriView -> LibriController
* Film -> FilmsView -> FilmsController
* Fruitore/Fruitori -> FruitoriView -> FruitoriController
* Prestito/Prestiti -> PrestitiView -> PrestitiController
* Storico -> StoricoView -> StoricoController

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*.

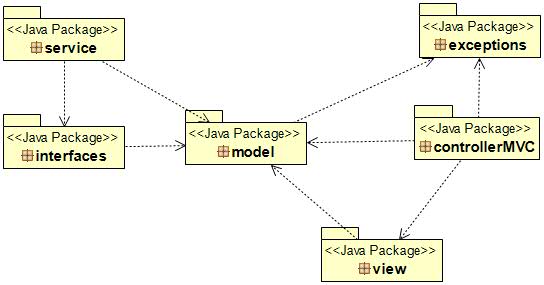
La mancanza della dipendenza da *Model* a *View* è confermata, a livello di codice, dal fatto che nessuna classe del package *Model* importi alcuna classe del package *View*.

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.

**UML PACKAGE MVC**



**UML PACKAGE**



# **UML MODEL-CONTROLLER**

# 

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

Caso d’uso “Registrazione”:

|  |  |
| --- | --- |
| 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 |

# **Operazioni di sistema**, rappresentate tramite *Diagramma di Sequenza di Sistema* (SSD):

UTENTE

SISTEMA

AddFruitore

msgChiediNome

Enter Nomee

msgChiediCognome

WHILE

Enter Cognome

! DataValida

Enter dataNascita

msgChiediDataNascita

IF

WHILE

msgChiediUsername

Utente Maggiorenne

! User

Disponibile

Enter Username

IsUsernameDisponibile

WHILE

msgChiediPSW1

! pswUguali

Enter PSW1

msgChiediPSW2

Enter PSW2

pswUguali

msgConfermaDati

Conferma dati

**CONTRATTI OPERAZONI DI SISTEMA:**  
riferite al caso d’uso principale

* *addFruiore*

**operazione**: addFruitore()

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**:

* *Enter Nome*

**operazione**: InputDati.leggiStringanonVuota(“Inserisci nome:”)

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**:

* *Enter Cognome*

**operazione**: InputDati.leggiStringaNonVuota(“Inserisci Cognome:”)

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**:

* *Enter dataNascita*

**operazione**: GestioneDate.creaDataGuidataPassata("inserisci la tua data di nascita: ", 1900)

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**: l’anno di nascita non è inferiore al 1900

* *Enter UserName*

**operazione**: InputDati.leggiStringaNonVuota(“Inserisci Username:”)

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**: l’username immesso non appartiene a nessun fruitore iscritto

* *Enter PSW1*

**operazione**: InputDati.leggiStringaNonVuota(“Inserisci la Password:”)

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**:

* *Enter PSW2*

**operazione**: InputDati.leggiStringaNonVuota(“Inserisci di nuovo la Password:”)

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**: la password immessa precedentemente coincide con quella digitata ora

* *Conferma dati*

**operazione**: InputDati.yesOrNo("Confermi l'iscrizione con questi dati?")

**Riferimenti**: caso d’uso: Registrazione

**Pre-condizioni**: è in corso l’iscrizione di un nuovo fruitore

**Post-condizioni**: i dati vengono confermati e viene creata un istanza di Fruitore

**UML di sequenza:**

GESTIONE DATE

INPUT DATI

FRUITORI VIEW

FRUITORI CONTROLLER

UTENTE

leggiStringa()

chiediNome()

addFruitore()

Nome

Nome

leggiStringa()

chiediCognome()

Cognome

Cognome

! DataValida

WHILE

creaDataGuidataPassata()

chiediDataNascita()

DataNascita

DataNascita

differenzaAnniDaOggi(DataNascita)

risposta

WHILE

IF

userNameDisponibilie ()=False

Risposta >= 18

leggiStringa()

chiediUsername()

FRUITORI

Username

Username

usernameDisponibile(Username)

risposta

IF

usernameNonDisponibile()

userNameDisponibilie()=False

WHILE

leggiStringa()

chiediPassword()

Password1

Password1 != Password2

Password1

confermaPassword()

leggiStringa()

Password2

Password2

confermaDati()

yesOrNo()

Risposta

Risposta

new()

FRUITORE

Risposta = true

IF

Caso d’uso “Richiedi Prestito”:

|  |  |
| --- | --- |
| NOME | **RICHIEDI UN PRESTITO** |
| ATTORE | FRUITORE |
| SCENARIO  PRINCIPALE | 1. <<include>> “**LOGIN FRUITORE”** (per sapere quale utente sta eseguendo la richiesta) 2. L’utente sceglie la funzionalità “Richiedi un prestito” 3. Il sistema chiede di che categoria sia la risorsa che si vuole prendere in prestito 4. Il sistema chiede all’utente se vuole visualizzare l’intero archivio e successivamente scegliere una risorsa o se preferisce filtrare l’archivio e successivamente scegliere una risposta 5. L’utente sceglie di visualizzare l’intero archivio 6. L’utente sceglie la risorsa fra quelle elencate 7. Il sistema stampa: “risorsa prenotata con successo”   FINE |
| SCENARIO  ALTERNATIVO | <<include>> “**RICERCA LIBRO”** (la ricerca avviene sui libri)  <<include>> ”**RICERCA FILM**” (la ricerca avviene sui film)  4a. (L’utente sceglie di filtrare la ricerca)  Il sistema mostra al fruitore i vari criteri per filtrare la ricerca (i criteri dipendono dal  genere di categoria che si è scelto)  4b. L’utente inserisce i valori atti alla ricerca  TORNO AL PUNTO 9 |

# **Operazioni di sistema**, rappresentate tramite *Diagramma di Sequenza di Sistema* (SSD):

SISTEMA

UTENTE

richiediPrestito(utenteLoggato)

# 

MsgChiediCategoria

Enter categoria

visualizzaRisorse(categoria)

Mostra risorse prestabili

scegliRisorsa()

prestitoFattibile

effettuaPrestito(utente, risorsa)

IF

Conferma

Prestito  
Fattibile

**CONTRATTI OPERAZONI DI SISTEMA**:  
riferite al caso d’uso principale e con l’utente che seleziona come categoria “Libri”)

* *richiediPrestito*

**Operazione**: richiediPrestito(utenteLoggato: Fruitore)

**Riferimenti**: caso d’uso: Richiedi prestito

**Pre-condizioni**: l’utente ha effettuato il Login

**Post-condizioni**: viene avviata la procedura di richiesta Prestito

* *Enter Categoria*

**Operazione**: InputDati.leggiStringa(“inserisci categoria:”)

**Riferimenti**: caso d’uso: Richiedi prestito

**Pre-condizioni**: è in corso la richiesta di un prestito

**Post-condizioni**: la stringa Categoria inserita è una categoria presente in archivio

* *visualizzaRisorse*

**Operazione**: visualizzaRisorse(categoria : String)

**Riferimenti**: caso d’uso: Richiedi prestito

**Pre-condizioni**: - è in corso la richiesta di un prestito

*- categoria* è una categoria valida

**Post-condizioni**: le risorse visualizzate sono disponibili al prestito

* *scegliRisorsa*

**Operazione**: selezionaLibro()

**Riferimenti**: caso d’uso: Richiedi prestito

**Pre-condizioni**: - è in corso la richiesta di un prestito

- Il fruitore può ricevere in prestito una risorsa della categoria selezionata

**Post-condizioni**: la risorsa selezionata ha copie disponibili

* *effetuaPrestito(utente, risorsa)*

**Operazione**: effettuaPrestito(utenteLoggato : Fruitore, risorsa Risorsa)

**Riferimenti**: caso d’uso: Richiedi prestito

**Pre-condizioni**: - utenteLoggato != null

- risorsa != null

- utenteLoggato non ha raggiunto il limite massimo di risorse

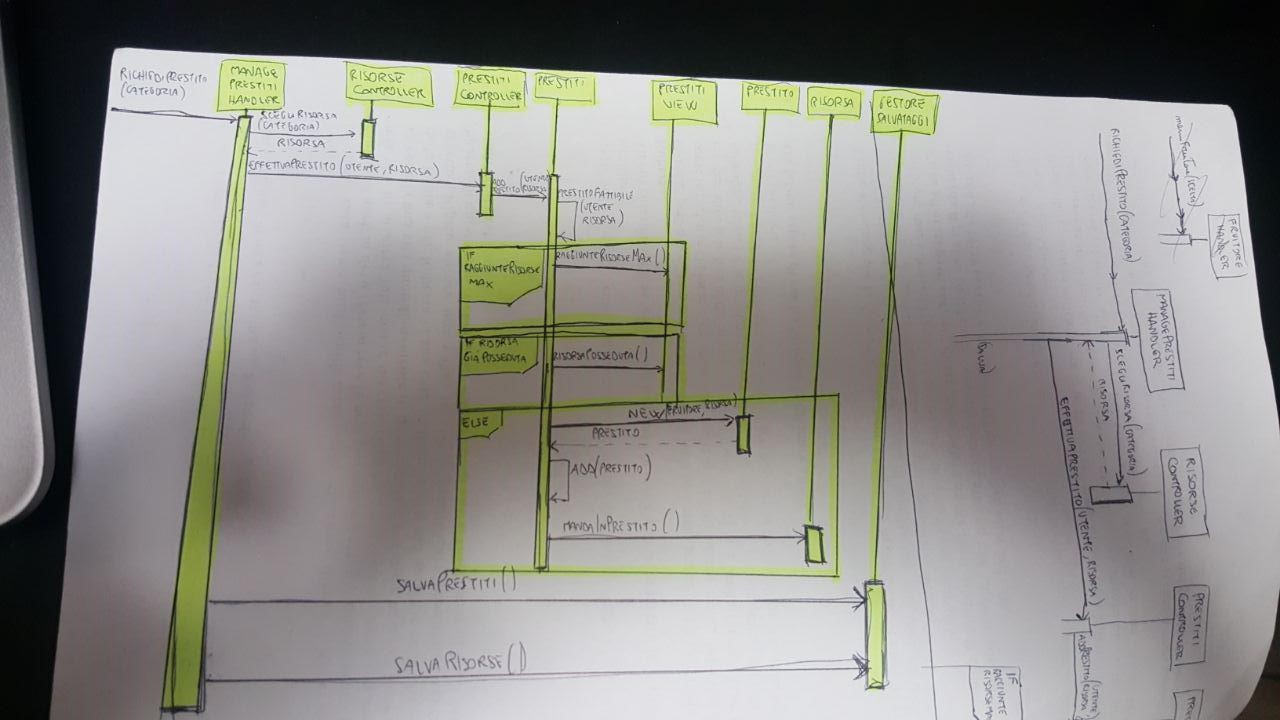
**Post-condizioni**: - creato prestito tra utenteLoggato e risorsa

- aggiunto prestito all’elenco dei prestiti

- il prestito non è già presente in archivio (*utente* non ha già *risorsa* in prestito)

**DIAGRAMMA DI SEQUENZA:**

fatto ma aspetterei a scriverlo qua, magari cambiano ancora cose applicando pattern.



Controller GRASP

Ogni interazione con utente è gestita da un menu, che viene stampato da un controller: sempre il controller riceve in input la selezione dell’utente e la manda all’Handler responsabile, che può coordinare l’operazione di sistema senza richiedere l’interazione con l’utente.

HANDLER sono controller dei casi d’uso:

ora diventati istanziabili

• Difetto di progettazione è avere un

controller gonfio di responsabilità

Ø Il Controller *delega*, non esegue!

# VIEW

Sarebbero da istanziare usando una interfaccia. -> aggiornare uml

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

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 possano esser 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, altrimenti era una variabile globale. Ora viene passata l’istanza dell’utente loggato tra i metodi.

Dividere i controller in più classi, se sono troppo gonfi (extract class)

Aggiunte ECCEZIONI PERSONALIZZATE in Prestiti.addPrestito(), vengono catturate da PrestitiController.effettuaPrestito()

**FATTO**:  
Ho modificato ancora un pochino Libri e Films, riuscendo a spostare altri metodi in Risorse come hai fatto te:   
Si potrebbero eliminare Libri e Films (gli è rimasto un solo metodo), lasciare tutto in Risorse (così forse se aggiungiamo un nuovo tipo di risorsa non c’è bisogno di modificare Risorse, il metodo add(Risorsa) andrebbe bene lo stesso), e in Archivio fare un vettore unico di risorse (come nel metodo tutteLeRisorse()), le risorse si distinguerebbero dall’ID.  
ANZI in realtà Archivio e Risorse sarebbero la stessa cosa, dato che anche Archivio ha solo due metodi utili.

Risorsa poi potrebbe diventare un’interfaccia come ci aveva suggerito la Zanella così possiamo dire che programmiamo orientati alle interfacce.

(gli errori che rimangono sono per l’oggetto dei salvataggi, non l’ho ancora fatto arrivare agli handler)

# PATTERN

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

FACTORY per creare oggetti che possono variare, tipo quelli che implementano interfaccia (gestoreSalvataggi, view(?))

PURE FABRICATION: gestoreSalvataggi, MenuFactory (?)

LibriController, FilmsController solo in RisorseController: la distinzione tra il tipo di risorse è presente solo qua, nel resto del programma si fa riferimento solo a RisorseController. Se si dovesse aggiungere una tipologia di risorsa bisognerà creare il suo controller e modificare solo RisorseController.

Gli handler gestiscono le scelte dei menu