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

**REFACTORING ARCHIVIO MULTIMEDIALE**

*RELAZIONE ATTIVITA’ SVOLTE*

Landi Federico

Prandini Stefano

Sommario

[**1) PRINCIPIO DI SEPARAZIONE MODELLO-VISTA** 3](#_Toc531793054)

[UML PACKAGE MVC 4](#_Toc531793056)

[UML PACKAGE 4](#_Toc531793057)

[UML CLASSI 5](#_Toc531793058)

[**2) OPERAZIONI DI SISTEMA (responsabilità e pattern GRASP)** 6](#_Toc531793060)

[Caso d’uso “Registrazione” 6](#_Toc531793061)

[Diagramma di Sequenza di Sistema (SSD) 7](#_Toc531793062)

[CONTRATTI OPERAZONI DI SISTEMA 8](#_Toc531793071)

[UML di sequenza 10](#_Toc531793072)

[PATTERN GRASP 11](#_Toc531793073)

[Caso d’uso “Richiedi Prestito” 12](#_Toc531793074)

[Diagramma di Sequenza di Sistema (SSD): 12](#_Toc531793075)

[CONTRATTI OPERAZONI DI SISTEMA 13](#_Toc531793084)

[UML di sequenza 14](#_Toc531793085)

[PATTERN GRASP 16](#_Toc531793086)

[**3) PATTERN SOLID** 17](#_Toc531793087)

[Single Responsibility Principle (SPR) 17](#_Toc531793088)

[Open-Closed Principle (OCP) 17](#_Toc531793089)

[Dependency Inversion Principle (DPI) 17](#_Toc531793090)

[VIEW 19](#_Toc531793091)

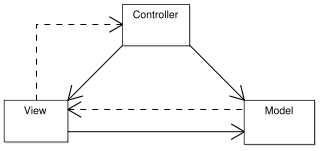
[MENU 19](#_Toc531793092)

[EVENTI/INTERAZIONI SISTEMA (da casi d’uso) per SSD 20](#_Toc531793093)

[REFACTOR 20](#_Toc531793094)

[PATTERN 21](#_Toc531793095)

# 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

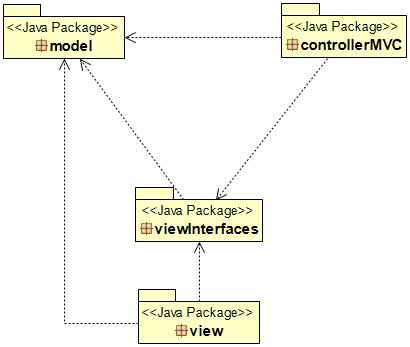
Ogni oggetto *Controller* possiede un attributo corrispondente al proprio *Model*, in modo da poter operare sui dati veri e propri dopo aver interagito con l’utente.  
Il *Controller* (collaborando con la *View*) interagisce con l’utente e modifica/aggiorna i dati del *Model*.

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*.

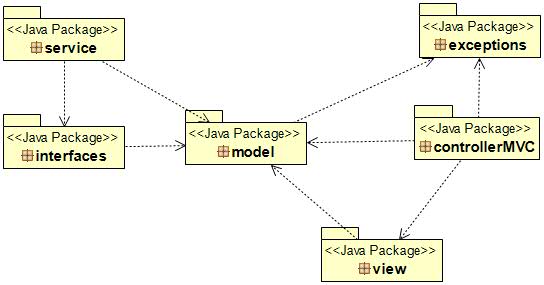
Ciò si vede anche dal diagramma UML dei package, nel quale la *View* dipende solo dal *Model*, il *Controller* dipende sia da *Model* che da *View* (dovendo interagire con entrambi) e il Model è indipendente da tutto il resto.

## UML PACKAGE MVC

Controller dipendono solo da interfacce della view ma non dalle classi che le implementano



## UML PACKAGE



## UML CLASSI

# 

# **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)

età

WHILE

IF

!userNameDisponibile ()

età >= 18

leggiStringa()

chiediUsername()

FRUITORI

Username

Username

usernameDisponibile(Username)

risposta

IF

usernameNonDisponibile()

!userNameDisponibile()

WHILE

leggiStringa()

chiediPassword()

Password1

Password1 != Password2

Password1

confermaPassword()

leggiStringa()

Password2

Password2

confermaDati()

yesOrNo()

Conferma

Conferma

Conferma= true

addFruitore(fruitore)

fruitore

new(datiFruitore)

FRUITORE

IF

### **PATTERN GRASP** per assegnazione responsabilità nel caso d’uso “Registrazione”:

* Information Expert: *assegnare responsabilità alla classe che possiede le informazioni necessarie per soddisfarla*
  + *Fruitori e FruitoriController* sono information expert per l’aggiunta di un fruitore

(hanno le informazioni su tutti i fruitori)

* + *GestoreSalvataggi* è information expert per il salvataggio dei dati aggiornati
* Creator*: assegnare responsabilità di creare nuove istanze di una classe*
  + *Fruitori* è responsabile della creazione di un nuovo fruitorie

(Contiene e aggrega oggetti di tipo fruitorie)

* Controller*: primo oggetto oltre lo strato UI che coordina operazioni di sistema*
  + *FruitoreHandler*: dopo che utente seleziona dal menu (UI) la voce “Registrazione”, coordina le operazioni di sistema per soddisfare la richiesta
* Pure Fabrication*: assegnare responsabilità coese ad una classe artificiosa*
  + *FruitoriController* è pure fabrication, per togliere a Fruitori le responsabilità di interazione con l’utente nell’aggiunta di un fruitore
  + *GestoreSalvataggi* è pure fabrication, per unificare i processi di salvataggio  
    (contro: le responsabilità non sono collocate insieme alle informazioni richieste)

*Controller GRASP -> HANDLER (sono controller dei casi d’uso)*

*Ogni interazione con l’utente è gestita da un menu: 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.*

## 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 6 |

### 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**: - utente != null

- risorsa != null

- utente non ha raggiunto il limite massimo di risorse

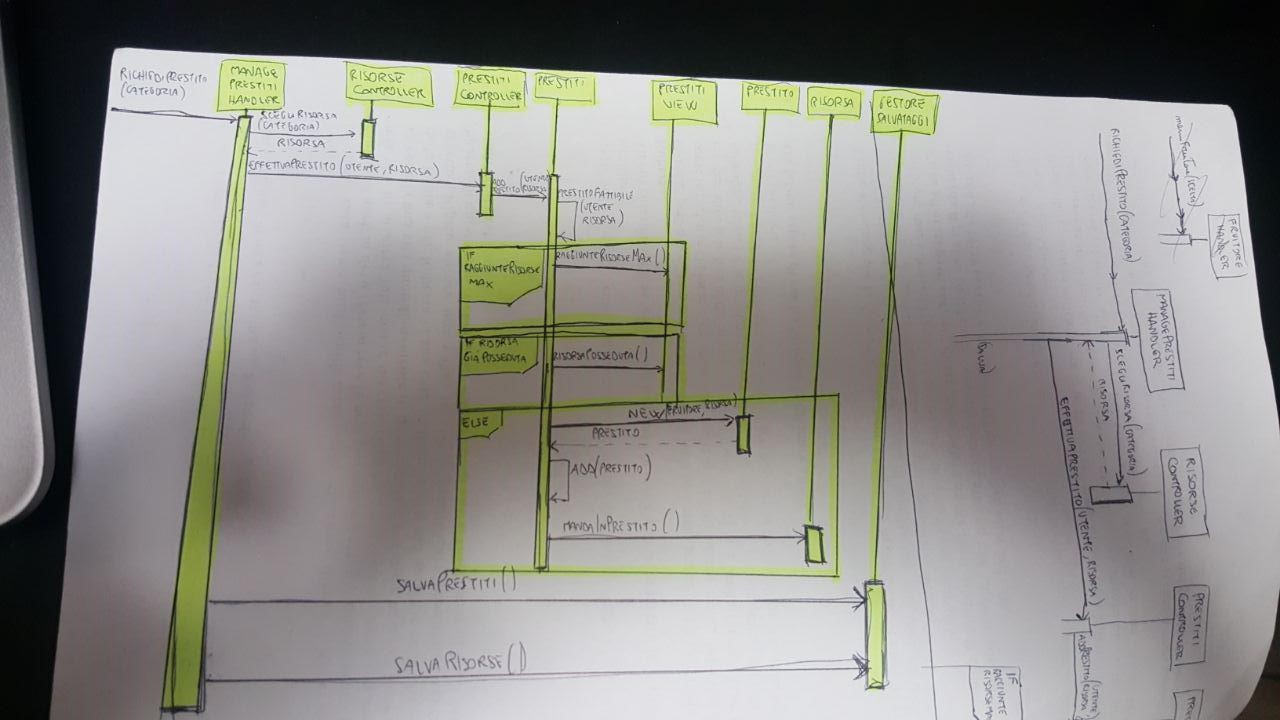
**Post-condizioni**: - creato prestito tra utente e risorsa

- 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.



### PATTERN GRASP per assegnazione responsabilità nel caso d’uso “richiedo prestito”:

* Information Expert: *assegnare responsabilità alla classe che possiede le informazioni necessarie per soddisfarla*
  + *RisorseController* è information expert per la scelta della risorsa del prestito   
    (ha le informazioni su tutte le risorse)
  + *Prestiti* e *PrestitiController* sono information expert per l’aggiunta di un prestito   
    (hanno le informazioni su tutti i prestiti)
  + *Risorsa* è information expert per l’aggiornamento del numero di copie in prestito   
    (è un suo attributo)
  + *GestoreSalvataggi* è information expert per il salvataggio dei dati aggiornati
* Creator*: assegnare responsabilità di creare nuove istanze di una classe*
  + *Prestiti* è responsabile della creazione di un nuovo prestito   
    (Contiene e aggrega oggetti di tipo Prestito)
* Controller*: primo oggetto oltre lo strato UI che coordina operazioni di sistema*
  + *PrestitiHandler:* dopo che utente seleziona dal menu (UI) la voce “Richiedi prestito”, coordina le operazioni di sistema per soddisfare la richiesta  
    (Handler rappresenta lo scenario del caso d’uso *richiedi prestito*)
* Pure Fabrication*: assegnare responsabilità coese ad una classe artificiosa*
  + *RisorseController* è pure fabrication, per togliere a *Risorse* le responsabilità di interazione con l’utente nella scelta di una risorsa
  + *PrestitiController* è pure fabrication, per togliere a *Prestiti* le responsabilità di interazione con l’utente nell’aggiunta di un prestito
  + *GestoreSalvataggi* è pure fabrication, per unificare i processi di salvataggio  
    (contro: le responsabilità non sono collocate insieme alle informazioni richieste)

*Controller GRASP -> HANDLER (sono controller dei casi d’uso)*

*Ogni interazione con l’utente è gestita da un menu: 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.*

# **3) PATTERN SOLID**

****

## Single Responsibility Principle (SPR)

Responsabilità = motivo per cambiare

Con le *View*:   
ogni view cambia solo se cambia la struttura degli oggetti che rappresenta (es. LibriView chiede i campi di Libro e li stampa);

Con i *Controller*:  
ogni controller cambia solo se cambia la struttura degli oggetti a cui è collegato (es. LibriController può cambiare solo se cambia la struttura di Libro);

Il *Main* prima poteva essere modificato per molti motivi, ora è più snello e delega gran parte di ciò che faceva prima (a GestoreSalvataggi e MainFacade);

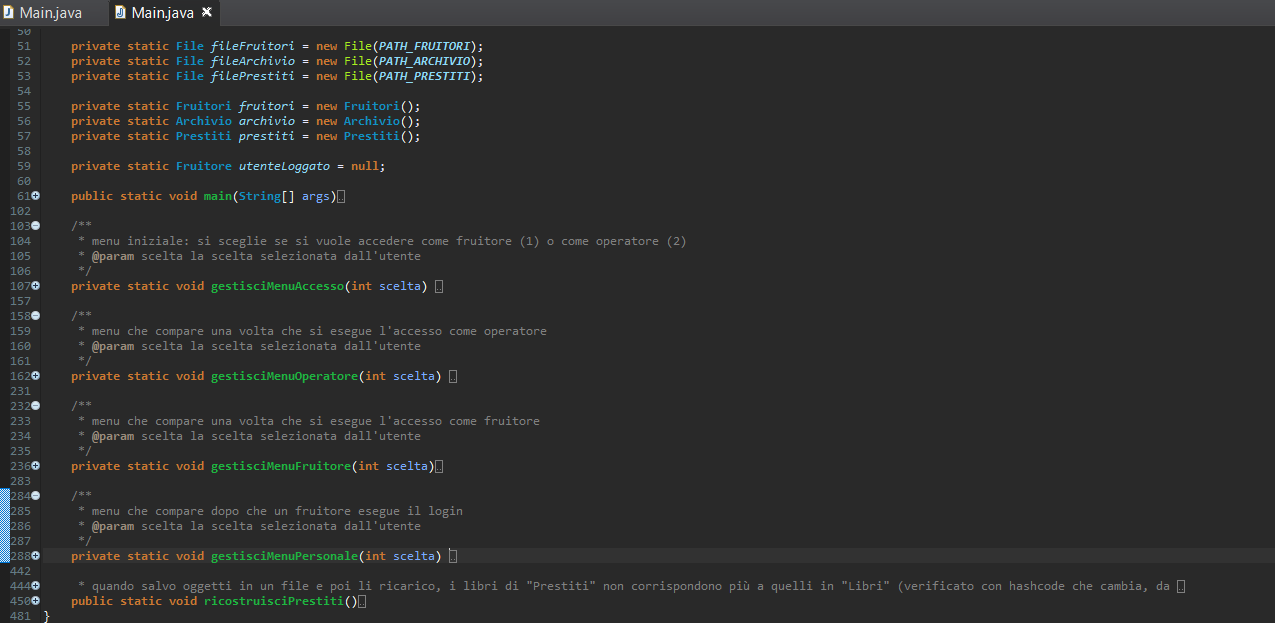
Le *varie classi* del progetto, prima del refactoring, potevano dover essere modificate per cambiamenti alla view oltre che al model (in quanto era tutto in un’unica classe): ora invece i cambiamenti che riguardano la gestione dei dati possono influenzare solo le classi del Model, mentre i cambiamenti relativi alla grafica influenzano solo le cassi della View;

I controller GRASP (*Handler*) possono cambiare solo se si decide di modificare la struttura del relativo menu (prima questa causa di cambiamento avrebbe comportato la modifica dell’unica classe che includeva sia model che view);

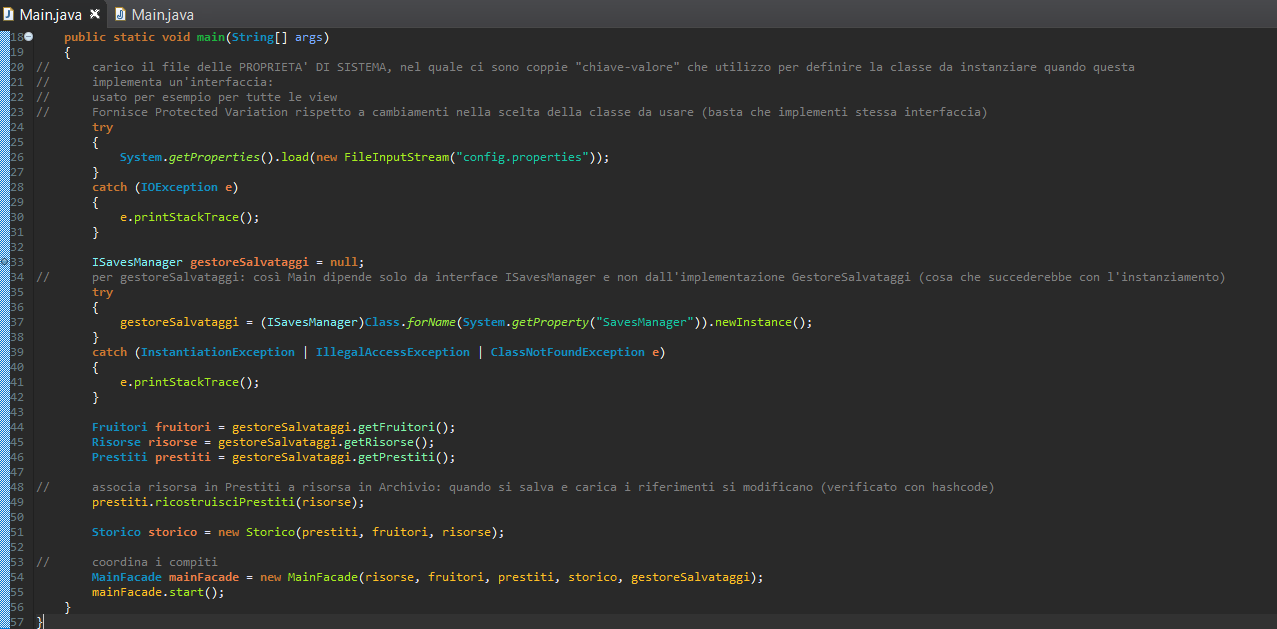
Ne risulta quindi che abbiamo separato 3 possibili cause di cambiamento delle classi “vecchie”, grazie alla separazione Model-View e all’introduzione degli Handler.

Esempio: codice del Main

Vecchio Main  
480 righe e tanti metodi diversi: poco snello e poco coeso.



Nuovo Main  
57 righe, unico compito quello di caricare i dati da file: il resto viene delegato.  
Snello e coeso!



## Open-Closed Principle (OCP)

Classi aperte all’estensione, chiuse alle modifiche

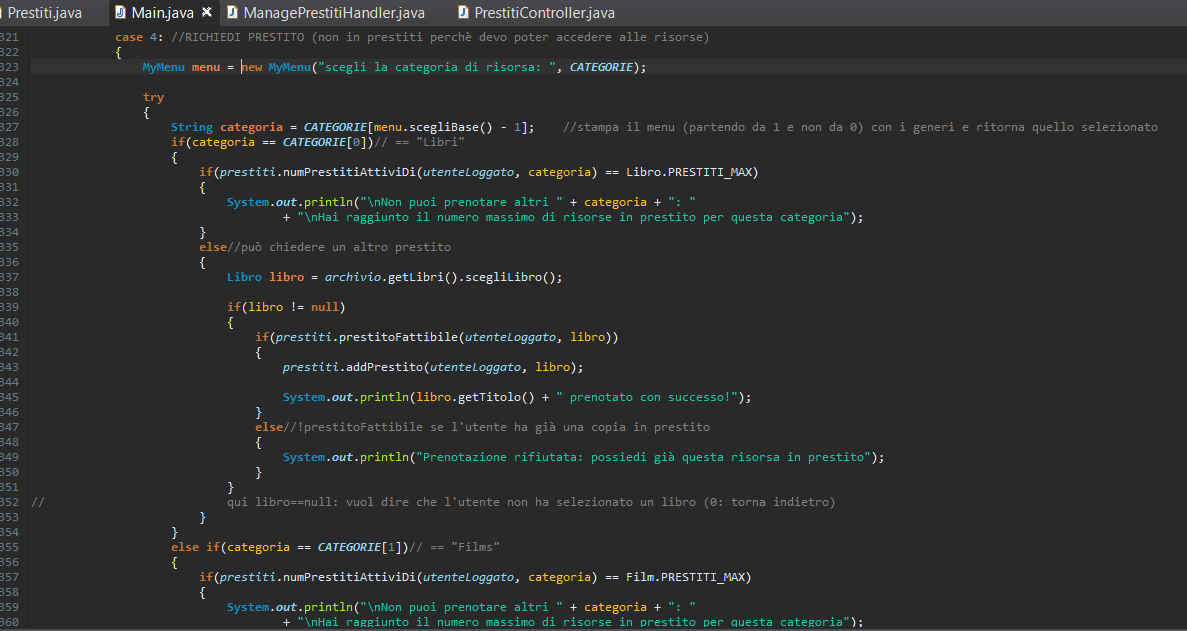
Abbiamo considerato come principale possibilità di cambiamento dei requisiti del progetto l’aggiunta di una nuova tipologia di Risorsa: per questo motivo abbiamo utilizzato in tutto il codice l’interfaccia *Risorsa*, in modo da non dover poi andare a cercare in tutte le classi i riferimenti alle risorse modificare il codice per utilizzare anche la nuova risorsa.  
Purtroppo alcuni metodi sono necessariamente diversi da risorsa a risorsa (es. Libri e Film hanno campi diversi): abbiamo contenuto questa “differenziazione” solamente nella classe RisorseController.

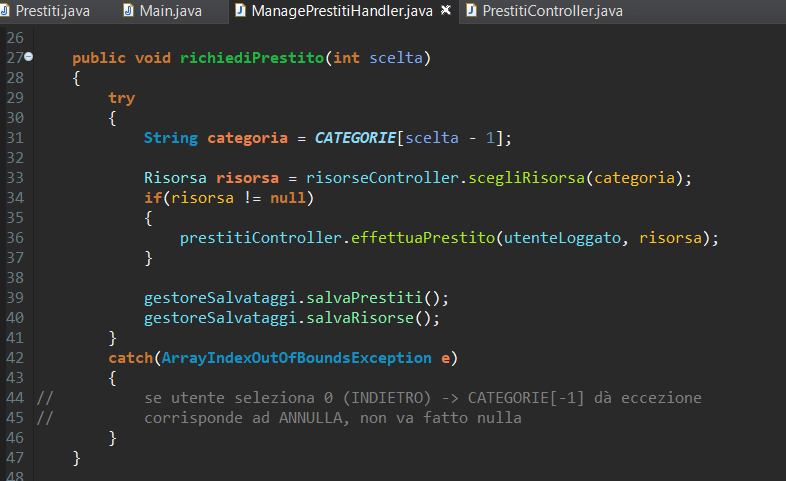
Se si dovesse quindi aggiungere una nuova tipologia di risorsa basterà crearne la classe, il relativo controller e la relativa view (apertura all’Estensione), e fare qualche piccola modifica alla classe RisorseController (inevitabile: nessun programma può essere chiuso al 100%).

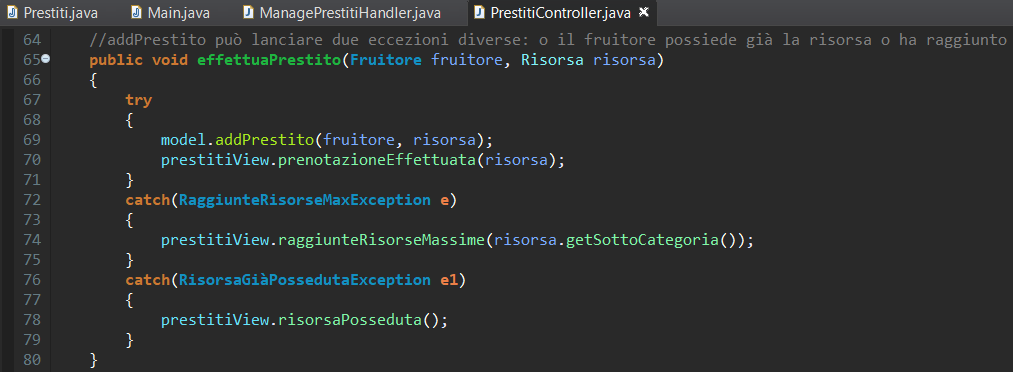
Inoltre, tutti gli attributi di tutte le classi sono *privati*: in questo modo tali attributi sono incapsulati e accessibili solamente tramite metodi getter e setter, mantenendo le classi che invocano questi metodi chiuse al cambiamento.

Esempio: codice per richiedere il prestito di una *Risorsa*

Caso “richiedi Prestito” vecchio  
Netta separazione, tramite uno Switch-Case, tra le diverse tipologie di Risorsa.  
Non aperto *all’estensione* ma alla *modifica*: se si aggiunge un tipo di risorsa bisogna modificare tutto il codice.



Caso “richiedi Prestito” nuovo  
Nessuna distinzione sul tipo di risorsa:   
il metodo “effettuaPrestito” accetta qualsiasi istanza di una classe che implementi l’interfaccia *Risorsa* (per ora solo Libro e Film, ma se si dovesse aggiungere una nuova tipologia di risorsa, basterà che tale classe implementi l’interfaccia *Risorsa*).  
(L’Handler riceve la “scelta” selezionata dall’utente)  
  
  




## 

## Dependency Inversion Principle (DPI)

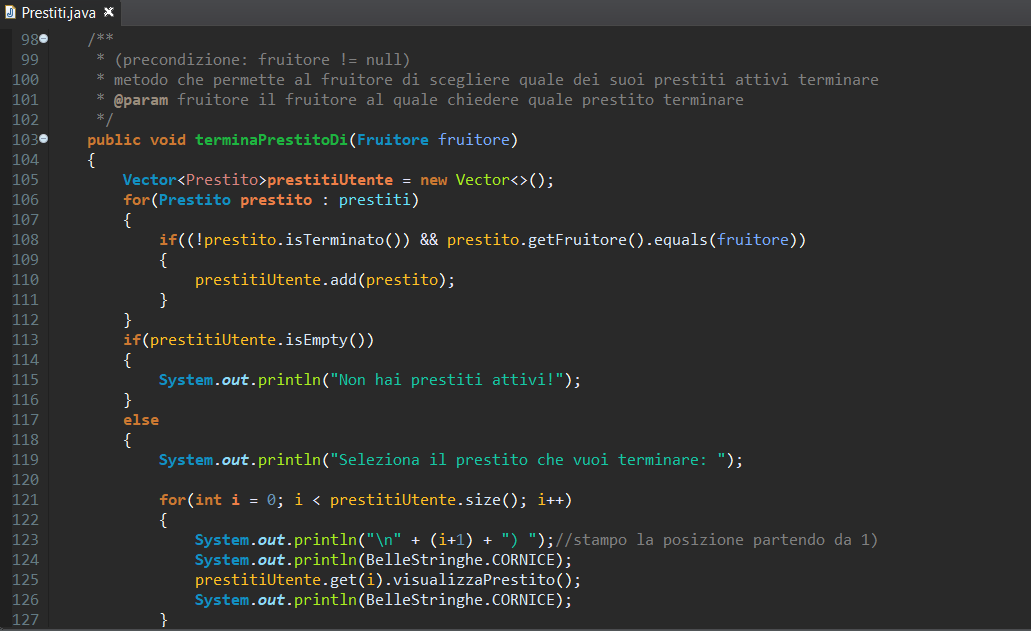
*I moduli di alto livello non devono dovrebbero dipendere da moduli di basso livello.*

Abbiamo applicato questo principio a tutte le classi View (che nella versione precedente del progetto nemmeno esistevano):   
All’inizio, nell’applicare il pattern MVC, le abbiamo create come classi statiche (in quanto servivano solo per stampare a console), successivamente le abbiamo trasformate in classi istanziabili facendo implementare ad ognuna di esse un’interfaccia.  
Infine abbiamo messo la loro istanziazione all’interno dei Controller MVC, facendo riferimento solo alle interfacce.

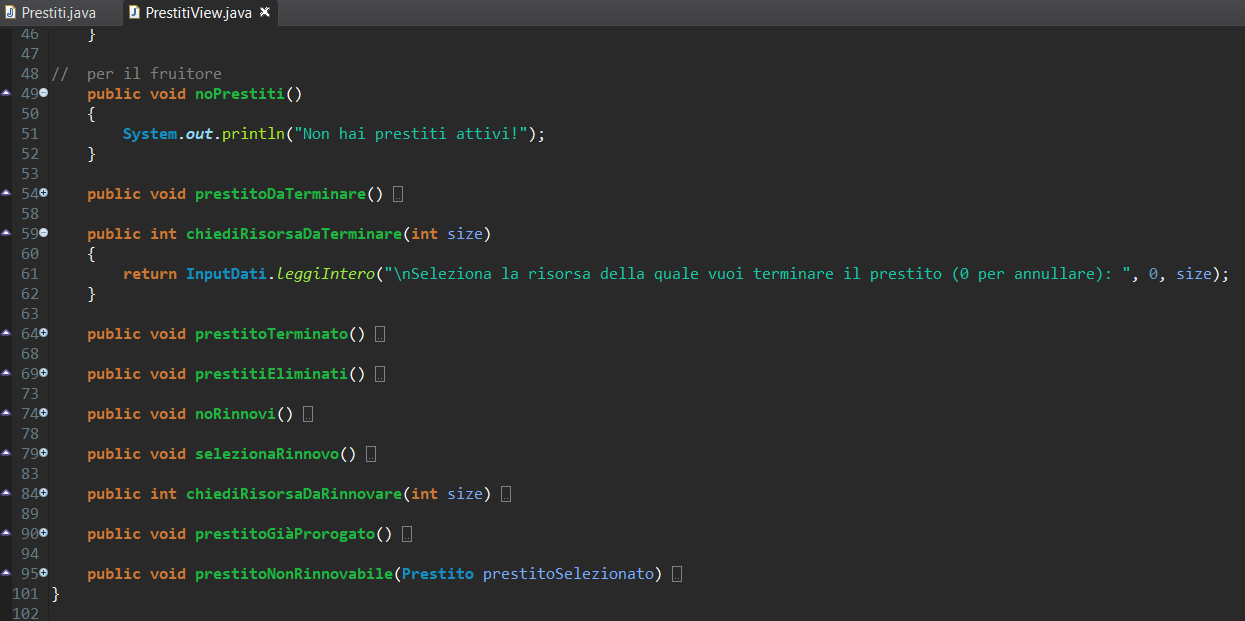
In questo modo, i Controller (moduli di “più alto livello”) non dipendono direttamente dalle View (moduli di “più basso livello”) ma dalle loro interfacce.

Anche il *Gestore dei salvataggi* viene definito tramite un’interfaccia (ISavesManager), in modo che se in una futura implementazione il progetto dovesse utilizzare una base di dati, il cambiamento sarà indolore.

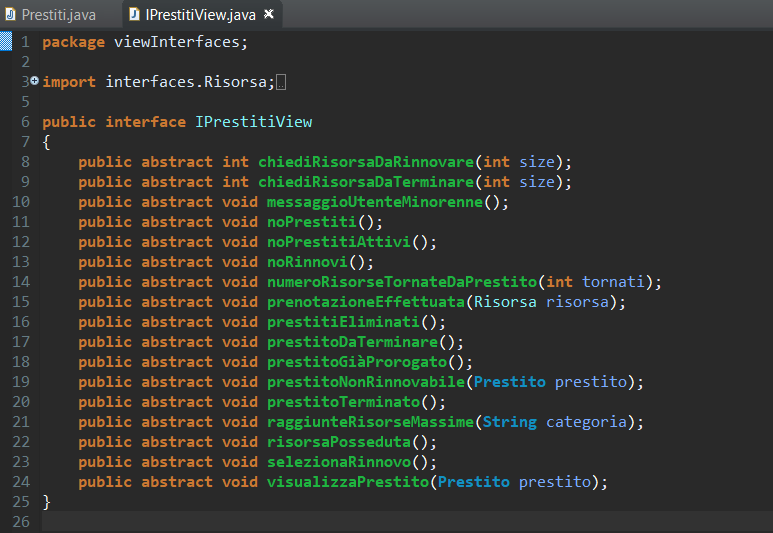
Progetto vecchio:  
  
 metodi per stampare a console “immersi” all’interno del codice



Progetto nuovo:  
  
Classe *View*



Interface *IView*



(esempio dell’implementazione delle view facendo riferimento solo alle *interfaces:* nel punto successivo)

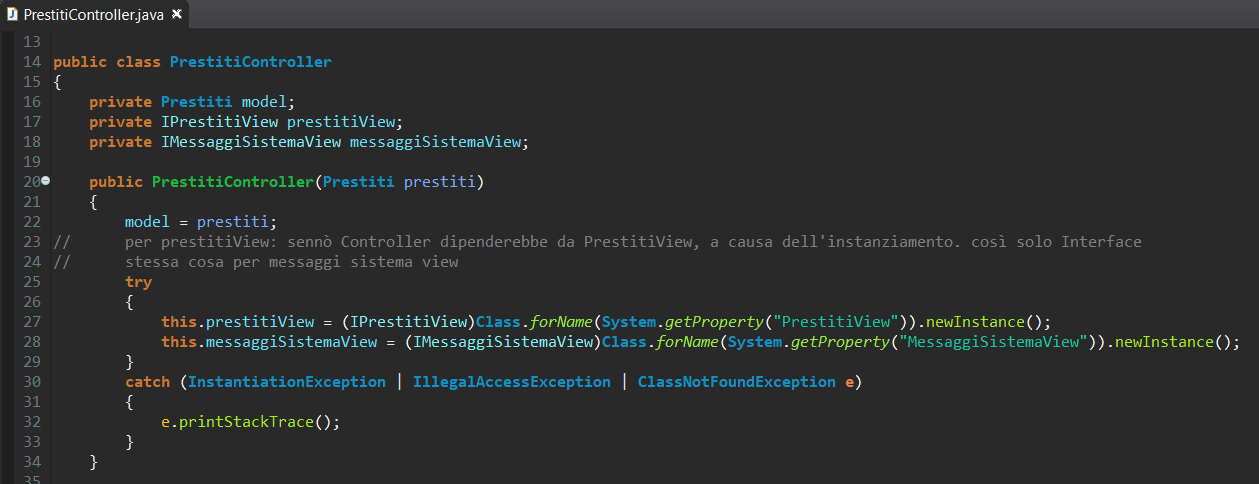
### Progettazione guidata dai dati

Il passo successivo è stato quello di eliminare completamente le dipendenze dei Controller dalle classi View (dipendenze che restavano per via delle istanziazioni), utilizzando un file di Proprietà di sistema (*config.properties*),che contiene un elenco di coppie chiave-valore.   
La classe da istanziare ora non viene definita nel codice (“*hard-coded”*) ma viene letta come proprietà dal file, in modo che la classe Controller (dove avviene questa istanziazione della view) non dipenda dalla classe da istanziare ma solo dall’interfaccia.  
Questo file viene caricato dal Main e “interrogato”, nei Controller, quando serve.

Anche il tipo di Gestore dei salvataggi viene letto da questo file di configurazione.

In questo modo assicuriamo anche Protected-Variation (GRASP) rispetto a cambiamenti nella scelta dell’”adattatore” da usare.

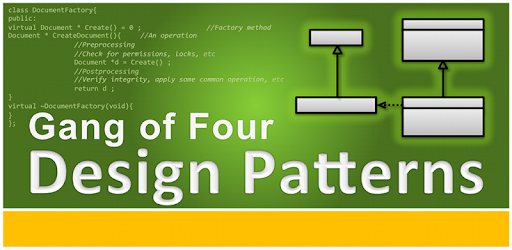
Progetto nuovo: istanziazione delle interfacce  
La classe controller ha dipendenza solo dall’interface *IPrestitiView*: il riferimento alla classe vera da istanziare è solamente nel file di proprietà di sistema.



File Config.properties:



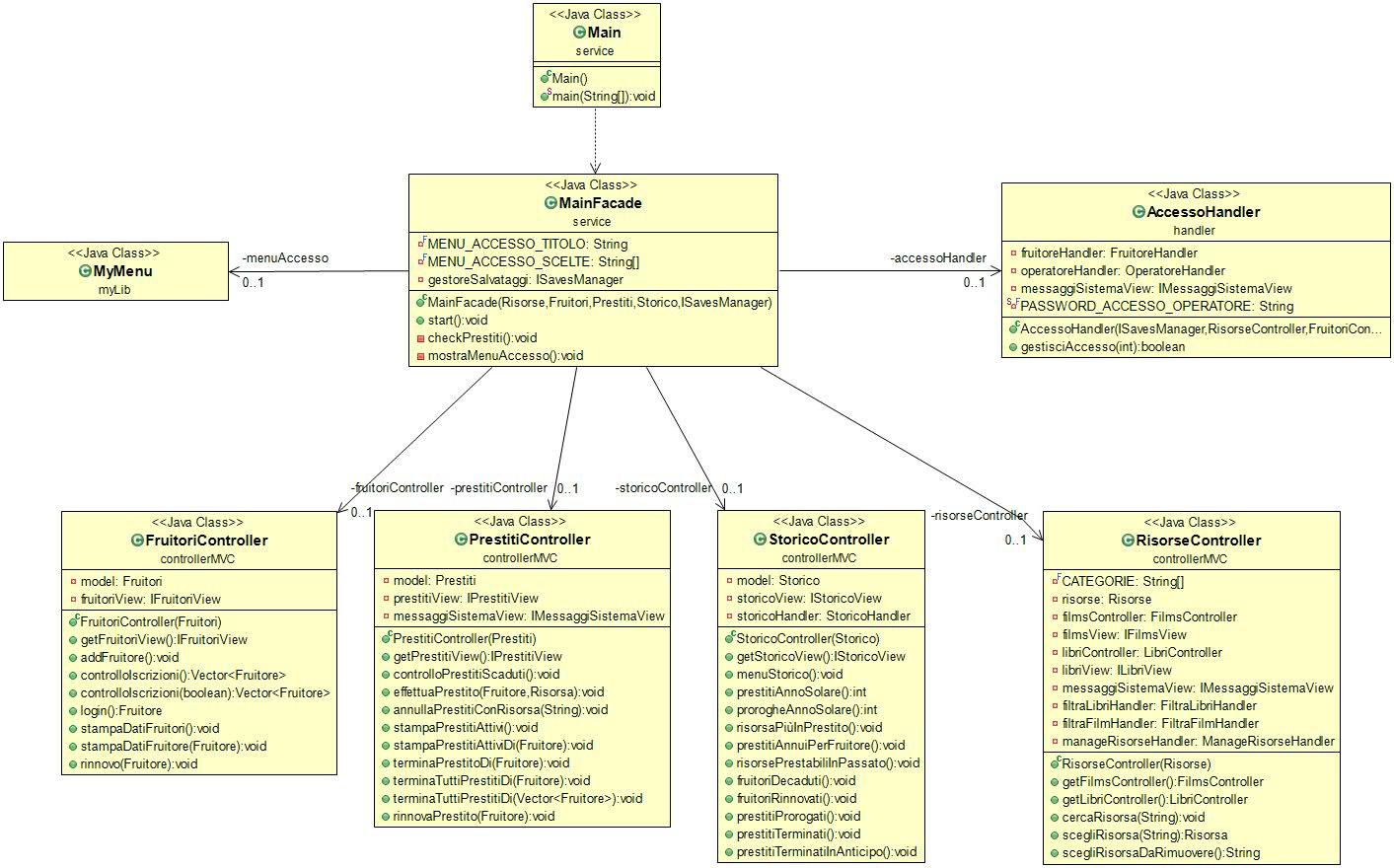
# **4) PATTERN Gang of Four (GoF)**

****

## **Facade**

Il Main non deve interagire con tutte le componenti del sottosistema. Delega la gestione delle funzionalità del sottosistema definendo un punto di contatto con esso, nel nostro caso la classe MainFacade: tale classe si occupa di coordinare la collaborazione tra i vari componenti del sistema, in particolare creando tutti i Controller e avviando la “parte grafica”, cioè la stampa del menu iniziale.

In questo modo l’accoppiamento tra il Main e il sottosistema risulta molto più basso.



*Il Main ha una dipendenza dal MainFacade, che crea i Controller, avvia il menu iniziale (di accesso) e delega a sua volta i compiti successivi ad AccessoHandler, che in base alla scelta dell’utente nel menu deciderà cosa fare.*

*Diagramma di sequenza (semplificato):*

AccessoHandler

s

MyMenu

MainFacade

Main

gestisciAccesso()

accessoHandler

New()

New()

menuAccesso

start()

New()

mainFacade

# **Factory**

# 

<<interface>>  
Risorsa

Film

Libro

RisorseController

RisorseFactory

Risorse risorse  
creaRisorsa  
(categoria)

* **Client** -> RisorseController
* **Factory** -> RisorseFactory
* **Abstract** -> Risorsa
* **Product1** -> Libro
* **Product2** -> Film

*RisorseFactory* crea l’oggetto richiesto da *RisorseController* (il client):   
*Risorsa* è l’interfaccia che implementano tutti gli oggetti che possono essere creati (nel nostro caso: *Libro* e *Film*).

Libro

RisorseFactory

RisorseController



menuAddRisorsa()

aggiungiRisorsa(categoria)

createRisorsa(categoria)

createLibro()

new()

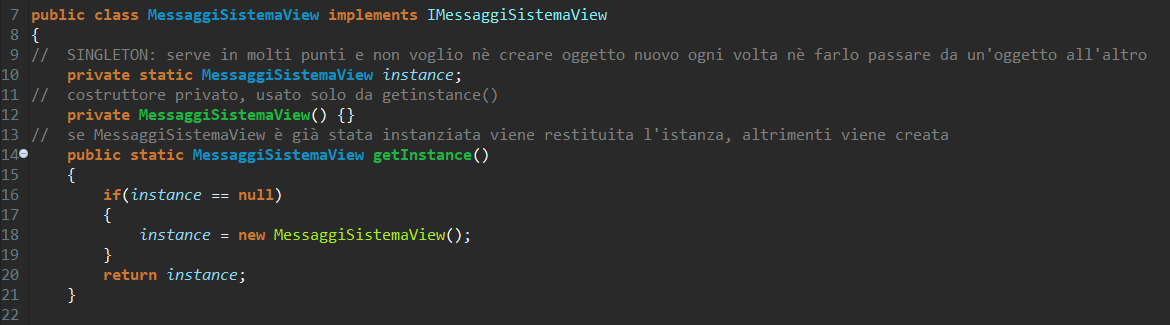


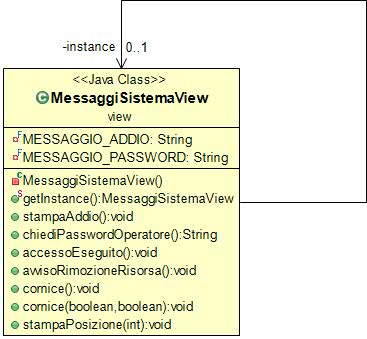
libro

## **Singleton**

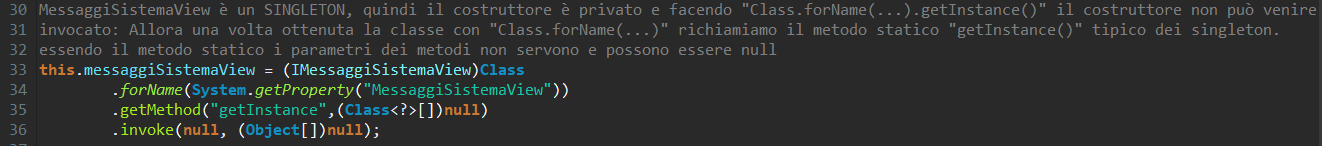
Abbiamo notato che la classe “*MessaggiSistemaView*” (che, come dice il nome, si occupa di stampare messaggi generici, come per esempio cornici o posizioni) veniva usata in molte classi: era quindi importante che tale classe non venisse istanziata in punti diversi.

La soluzione è stata quella di rendere tale classe un Singleton: pattern GoF che assicura l’esistenza di una sola istanza di una classe, fornendo un punto di accesso globale ad essa (il metodo getInstance() ).





Facendo ciò è però sorto un problema, legato al punto precedente, cioè l’istanziare gli oggetti View usando proprietà di sistema: il metodo newInstance() invoca infatti il costruttore della classe, che nel caso del Singleton non andrebbe usato. Abbiamo quindi cambiato strategia di istanziazione, per fare in modo di invocare il metodo getInstace().  
In particolare, in PrestitiController e RisorseController:



## **Command**

Pattern:

Concrete Command

Client

Invoker

ICommand

Receiver

Implementazione:

gestisciAccesso()

AccessoHandler

MainFacade

ICommand

Fruitore Handler

AccessoFruitore Command

Operatore Handler

AccessoOperatore Command

* Client: MainFacade
* Invoker: AccessoHandler
* Receiver: FruitoreHander/OperatoreHandler

In AccessoHandler, se l’utente indica di essere un fruitore o un operatore del sistema, va in ogni caso gestito l’accesso, compito di FruitoreHandler o OperatoreHandler.   
Per disaccoppiare il richiedente del servizio (AccessoHandler) dall’esecutore (FruitoreHandler/OperatoreHandler) e per definire meglio le operazioni da eseguire, abbiamo deciso di separare tali operazioni in due classi Command (una per i comandi da eseguire in caso l’utente sia un fruitore, una per quelli da eseguire in caso sia un operatore). Queste due classi Command implementano un’interfaccia comune ICommand, che contiene il metodo gestisciAccesso().

In questo modo abbiamo ridotto l’accoppiamento richiedente/esecutore e abbiamo migliorato l’estendibilità: risulta infatti più semplice, ora, aggiungere nuovi eventuali comandi.

# **5) REFACTORING**

# 

# 

Il package “testing” contiene i vari test che abbiamo eseguito ad ogni modifica del codice per verificare che esso funzionasse sempre correttamente e che i cambiamenti non comportassero malfunzionamenti.

(La maggior parte dei passaggi di refactoring sono avvenuti nello svolgimento di questa “seconda parte” di progetto, quindi non sono visibili in modo evidente confrontando questa versione del codice con la Parte5 della “prima parte” del progetto.)

Il refactoring più evidente riguarda l’archiviazione delle risorse:

# VIEW

Sarebbero da istanziare usando una interfaccia. -> aggiornare UML

Si potrà creare una specie di controller grasp per creare e gestire tutti i controller?

# 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)

# 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

Nei vari passaggi di questo progetto ci sono stati molti tipi di refactoring che non sono tangibili, quindi alcune cose non sono rispetto alla parte 5.

**risorseController molto grosso**

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.