

Progetto di Ingegneria del Software

Parte 2

a.a. 2017/2018

REFACTORING ARCHIVIO MULTIMEDIALE



RELAZIONE ATTIVITA' SVOLTE

Prandini Stefano (712731)

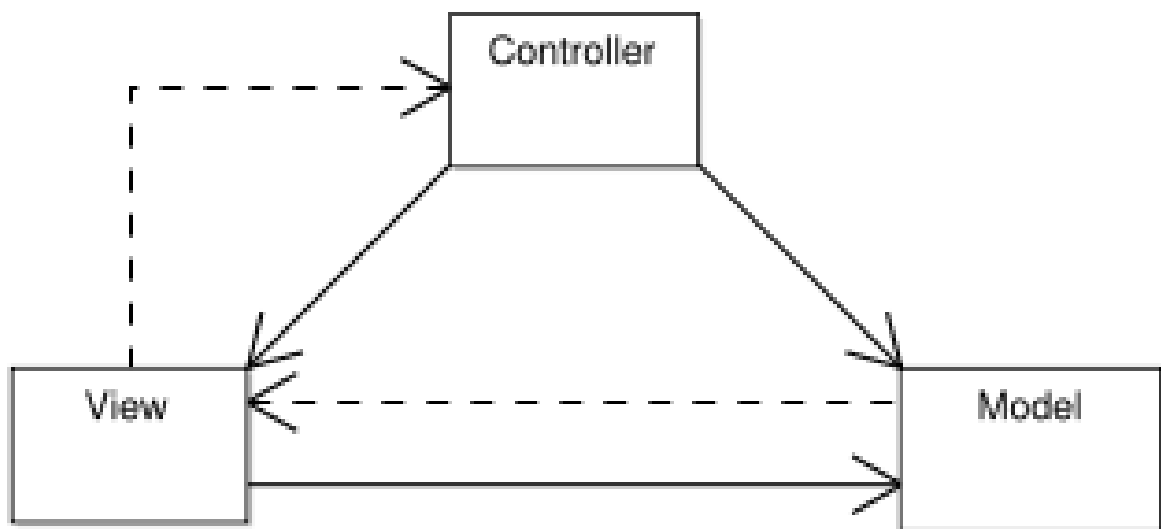
Landi Federico (713227)

Sommario

1) PRINCIPIO DI SEPARAZIONE MODELLO-VISTA.....	3
UML PACKAGE MVC	5
UML PACKAGE	5
UML CLASSI.....	6
2) OPERAZIONI DI SISTEMA (responsabilità e pattern GRASP).....	7
Caso d'uso "Registrazione"	8
Operazioni di sistema, rappresentate tramite SSD	9
CONTRATTI OPERAZIONI DI SISTEMA.....	10
UML di sequenza	12
PATTERN GRASP per assegnazione responsabilità.....	13
Caso d'uso "Richiedi Prestito"	14
Operazioni di sistema, rappresentate tramite SSD	14
CONTRATTI OPERAZIONI DI SISTEMA.....	15
UML di sequenza	16
PATTERN GRASP per assegnazione responsabilità.....	17
3) PATTERN SOLID	18
Single Responsibility Principle (SPR).....	19
Open-Closed Principle (OCP)	22
Dependency Inversion Principle (DPI)	25
Progettazione guidata dai dati	27
4) Gang of Four (GoF) Patterns.....	28
Facade.....	29
Factory	30
Singleton	31
Command	32
5) REFACTORING	33
Per sistemare metodi troppo lunghi:	35
EXTRACT METHOD.....	35
MOVE METHOD	36
EXTRACT CLASS.....	36
Per semplificare statements condizionali:	37
INTRODUCE ASSERTIONS.....	37
Per rendere le chiamate ai metodi più semplici.....	38
REPLACE CONSTRUCTOR WITH FACTORY METHOD.....	38

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 applicato 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.

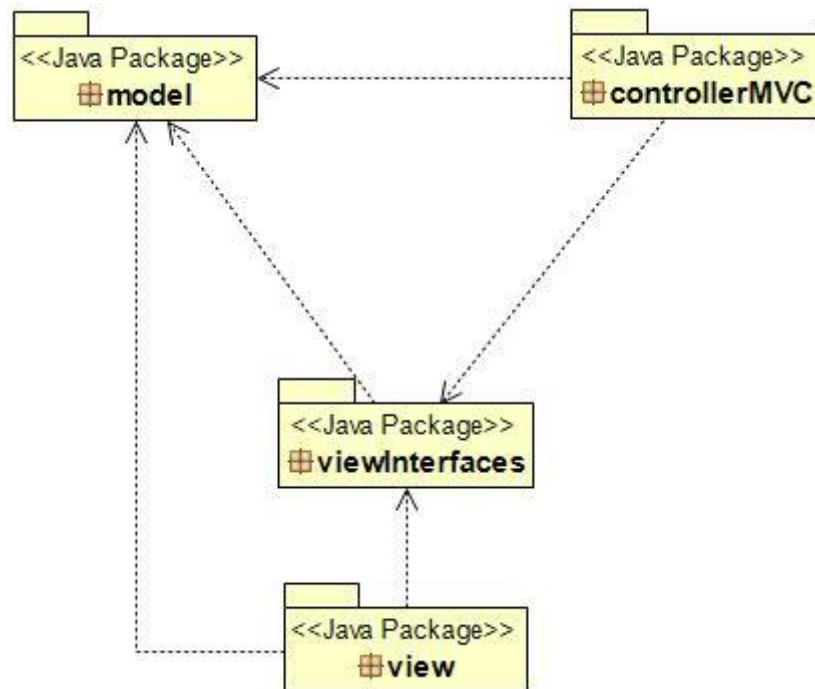
Il *Controller* (collaborando con la *View*) interagisce con l'utente e modifica/aggiorna i dati del *Model*.

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

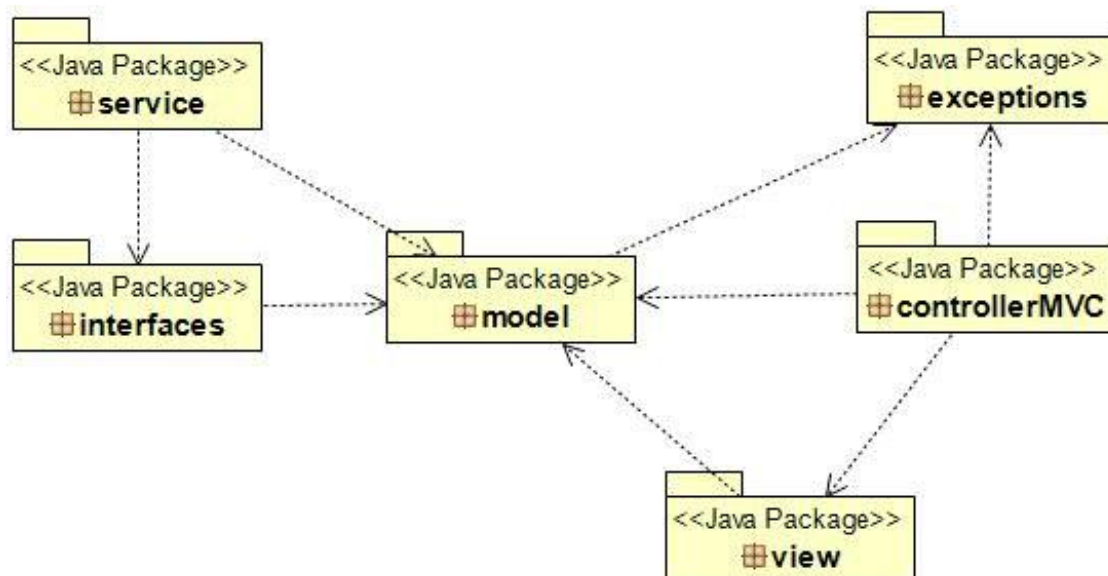
Ciò si vede anche dal diagramma UML dei package, nel quale la *View* dipende solo dal *Model*, il *Controller* dipende sia da *Model* che da *View* (dovendo interagire con entrambi) e il *Model* è indipendente da tutto il resto.

UML PACKAGE MVC

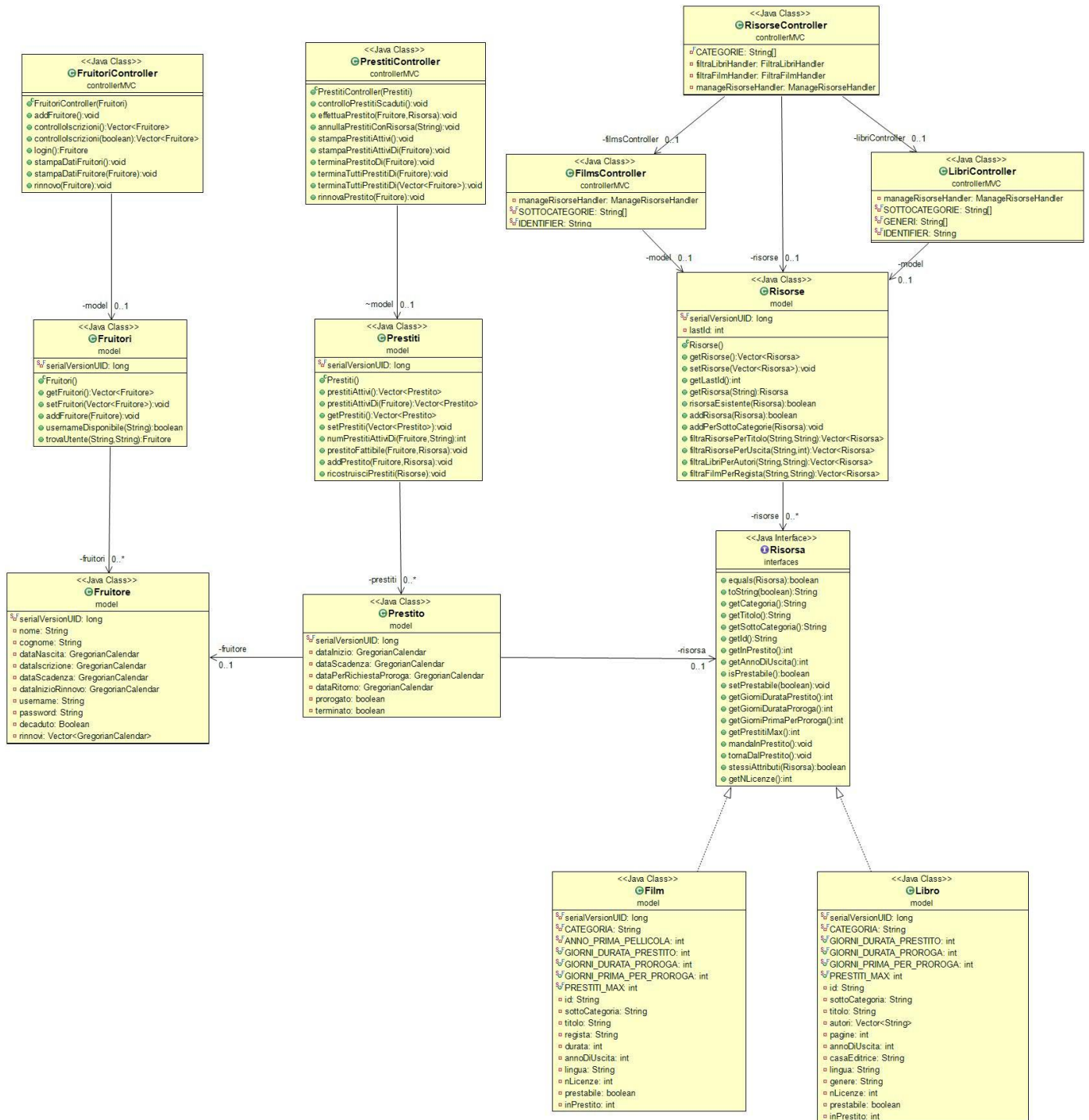
Controller dipendono solo da interfacce della view ma non dalle classi che le implementano



UML PACKAGE



UML CLASSI



2)

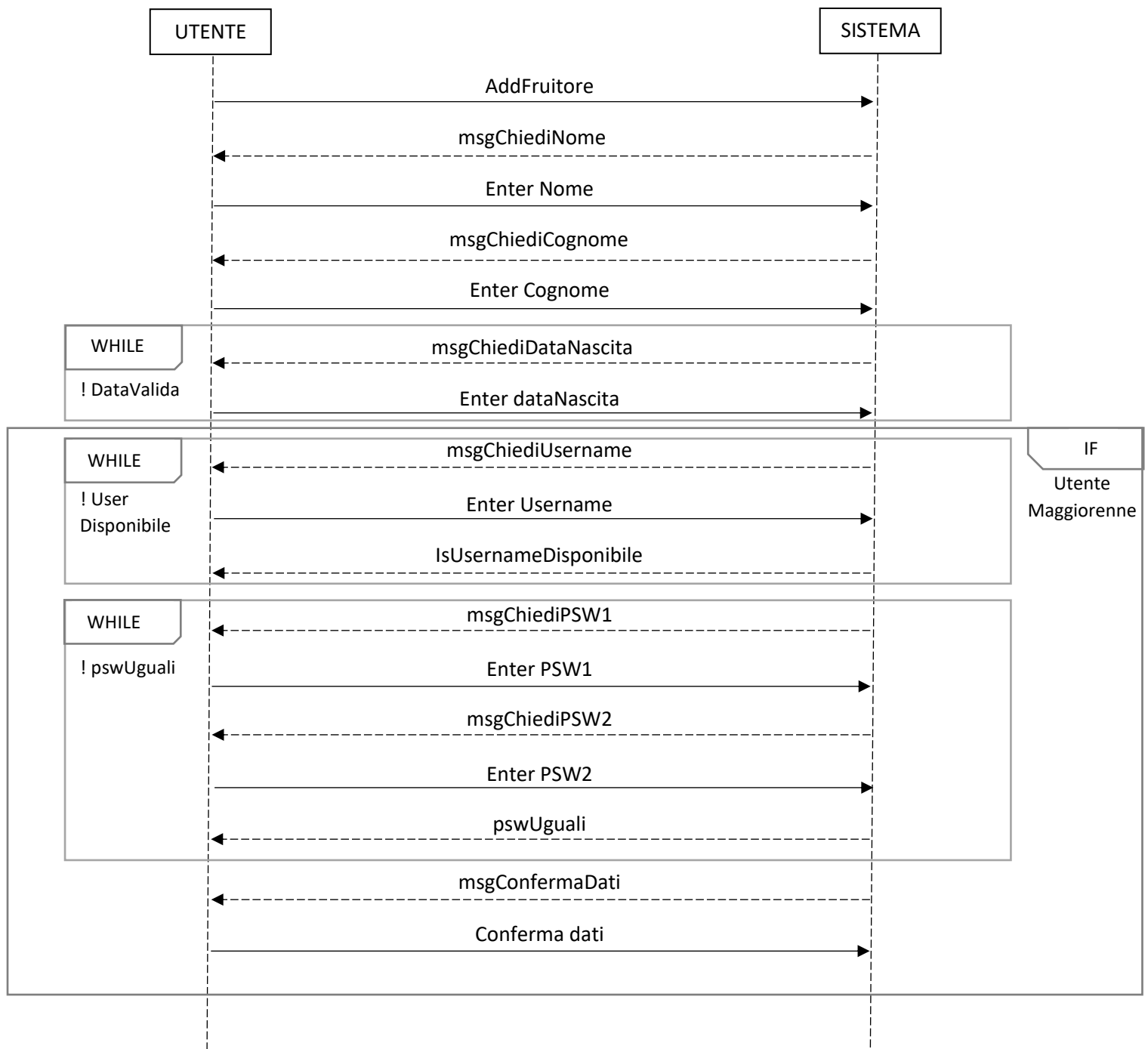
OPERAZIONI DI SISTEMA (responsabilità e pattern GRASP)



Caso d'uso "Registrazione"

NOME	REGISTRAZIONE
ATTORE	FRUITORE
SCENARIO PRINCIPALE	<ol style="list-style-type: none">1. L'utente sceglie la funzionalità "Registrazione"2. Il sistema presenta i campi anagrafici da compilare3. L'utente inserisce i dati anagrafici4. Il sistema presenta il campo Username5. L'utente inserisce il dato6. Il sistema presenta due volte il campo Password7. L'utente completa i due campi8. Il sistema stampa "Registrazione avvenuta con successo" FINE
SCENARIO ALTERNATIVO	4a. (<u>Precondizione: la data inserita non esiste o è futura</u>) Il sistema mostra un messaggio di errore e chiede di reinserire la data TORNA AL PUNTO 2
SCENARIO ALTERNATIVO	4a. (<u>Precondizione: l'utente non è maggiorenne</u>) Il sistema informa l'utente che non può registrarsi (e termina la procedura di registrazione) FINE
SCENARIO ALTERNATIVO	6a. (<u>Precondizione: username non disponibile</u>) Il sistema indica all'utente di scegliere un altro username TORNA AL PUNTO 4
SCENARIO ALTERNATIVO	4a. (<u>Precondizione: le 2 password non coincidono</u>) 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):



CONTRATTI OPERAZIONI DI SISTEMA:

riferite al caso d'uso principale

- *addFruitore*

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

- *Enter Cognome*

operazione: InputDati leggiStringaNonVuota("Inserisci Cognome:")

Riferimenti: caso d'uso: Registrazione

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

- *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

- *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*
-

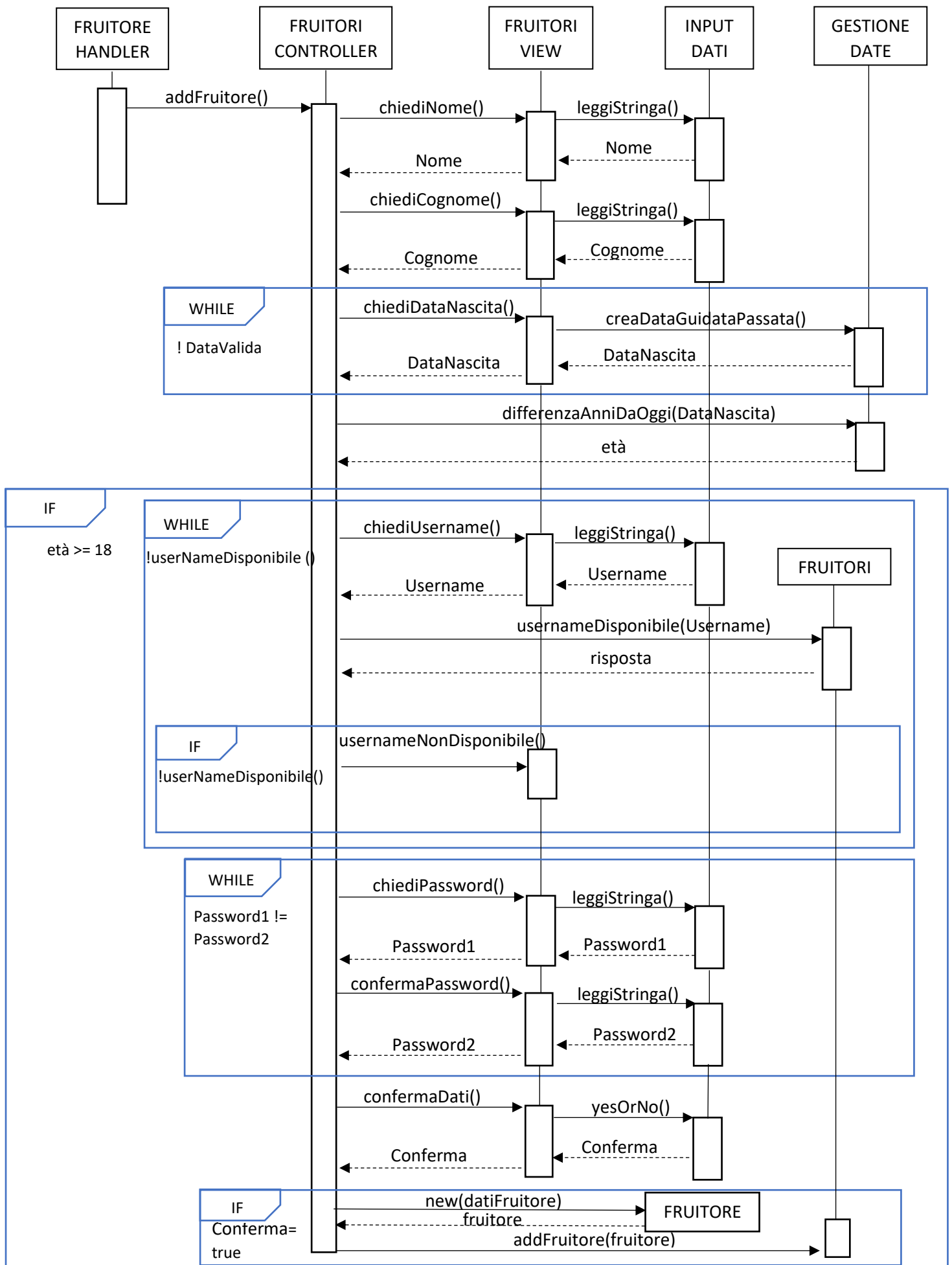
operazione: InputDati.yesOrNo("Confermi l'iscrizione con questi dati?")

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:



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 fruitore (Contiene e aggrega oggetti di tipo fruitore)
- 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)

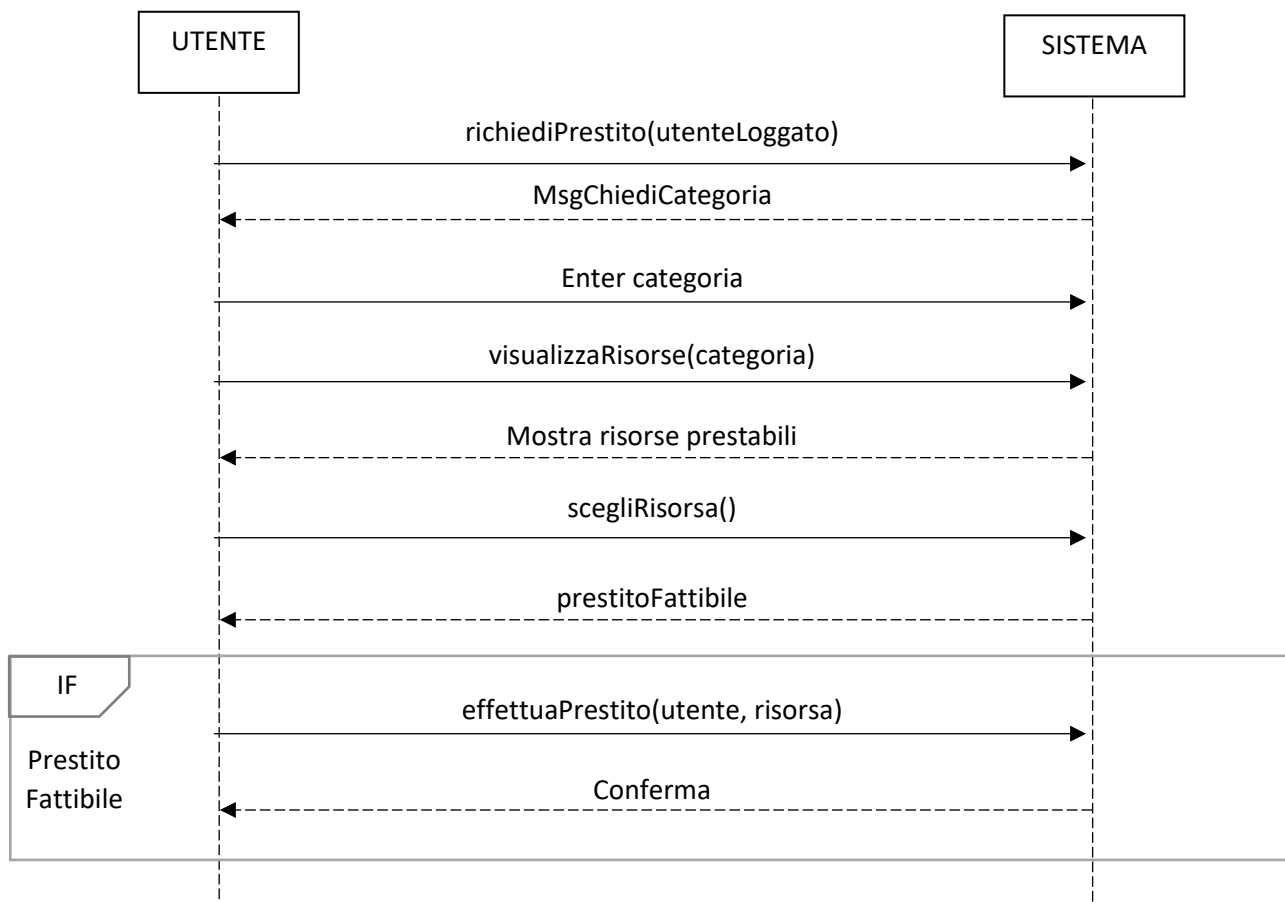
Controller GRASP -> HANDLER (sono controller dei casi d'uso)

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	<ol style="list-style-type: none"> 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" <p>FINE</p>
SCENARIO ALTERNATIVO	<p><<include>> "RICERCA LIBRO" (la ricerca avviene sui libri) <<include>> "RICERCA FILM" (la ricerca avviene sui film)</p> <p>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)</p> <p>4b. L'utente inserisce i valori atti alla ricerca</p> <p>TORNO AL PUNTO 6</p>

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



CONTRATTI OPERAZIONI DI SISTEMA:

referite 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

- *effettuaPrestito*(utente, risorsa)

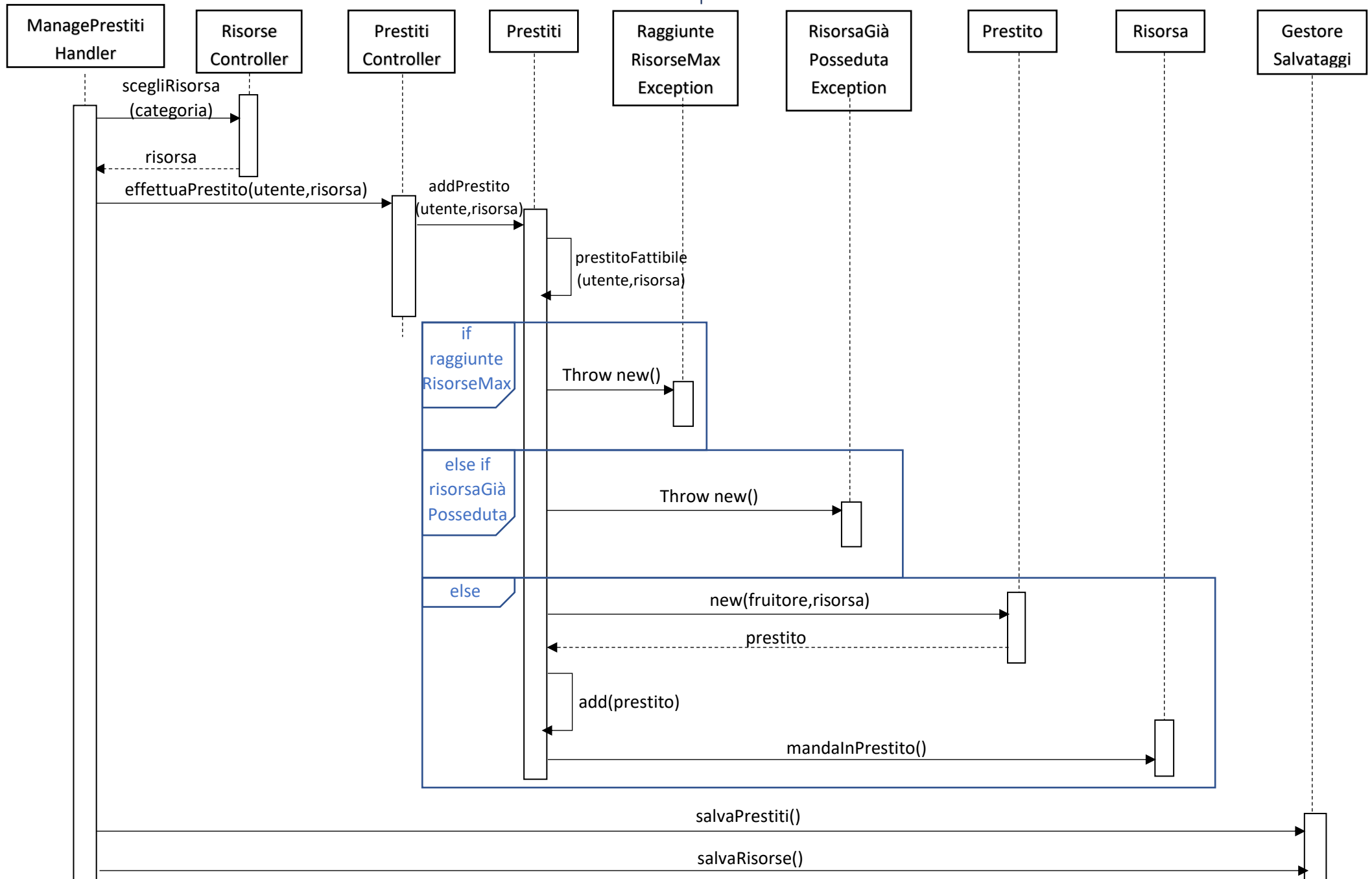
Operazione: *effettuaPrestito*(utenteLoggato : Fruitore, risorsa Risorsa)

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)

UML di sequenza:



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



SOLID principles

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);

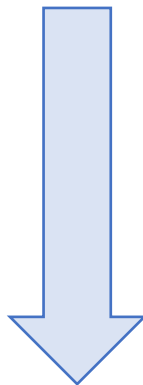
Il *Main* prima poteva essere modificato per molti motivi, ora è più snello e delega gran parte di ciò che faceva prima (a GestoreSalvataggi e MainFacade);

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 classi 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.

```
Main.java Main.java x
50
51 private static File fileFruitori = new File(PATH_FRUITORI);
52 private static File fileArchivio = new File(PATH_ARCHIVIO);
53 private static File filePrestiti = new File(PATH_PRESTITI);
54
55 private static Fruitori fruitori = new Fruitori();
56 private static Archivio archivio = new Archivio();
57 private static Prestiti prestiti = new Prestiti();
58
59 private static Fruitore utenteLoggato = null;
60
61+ public static void main(String[] args) {
102
103- /**
104  * menu iniziale: si sceglie se si vuole accedere come fruitore (1) o come operatore (2)
105  * @param scelta la scelta selezionata dall'utente
106  */
107+ private static void gestisciMenuAccesso(int scelta) {
157
158- /**
159  * menu che compare una volta che si esegue l'accesso come operatore
160  * @param scelta la scelta selezionata dall'utente
161  */
162+ private static void gestisciMenuOperatore(int scelta) {
231
232- /**
233  * menu che compare una volta che si esegue l'accesso come fruitore
234  * @param scelta la scelta selezionata dall'utente
235  */
236+ private static void gestisciMenuFruitore(int scelta) {
283
284- /**
285  * menu che compare dopo che un fruitore esegue il login
286  * @param scelta la scelta selezionata dall'utente
287  */
288+ private static void gestisciMenuPersonale(int scelta) {
442
444+ * quando salvo oggetti in un file e poi li ricarico, i libri di "Prestiti" non corrispondono più a quelli in "Libri" (verificato con hashCode che cambia, da
450+ public static void ricostruisciPrestiti() {
481 }
```

Nuovo Main

57 righe, unico compito quello di caricare i dati da file: il resto viene delegato.

Snello e coeso!

```
Mainjava x Mainjava
18 public static void main(String[] args)
19 {
20     // carico il file delle PROPRIETA' DI SISTEMA, nel quale ci sono coppie "chiave-valore" che utilizzo per definire la classe da instanziare quando questa
21     // implementa un'interfaccia:
22     // usato per esempio per tutte le view
23     // Fornisce Protected Variation rispetto a cambiamenti nella scelta della classe da usare (basta che implementi stessa interfaccia)
24     try
25     {
26         System.getProperties().load(new FileInputStream("config.properties"));
27     }
28     catch (IOException e)
29     {
30         e.printStackTrace();
31     }
32
33     ISavesManager gestoreSalvataggi = null;
34     // per gestoreSalvataggi: così Main dipende solo da interface ISavesManager e non dall'implementazione GestoreSalvataggi (cosa che succederebbe con l'instanziamento)
35     try
36     {
37         gestoreSalvataggi = (ISavesManager)Class.forName(System.getProperty("SavesManager")).newInstance();
38     }
39     catch (InstantiationException | IllegalAccessException | ClassNotFoundException e)
40     {
41         e.printStackTrace();
42     }
43
44     Fruitori fruitori = gestoreSalvataggi.getFruitori();
45     Risorse risorse = gestoreSalvataggi.getRisorse();
46     Prestiti prestiti = gestoreSalvataggi.getPrestiti();
47
48     // associa risorsa in Prestiti a risorsa in Archivio: quando si salva e carica i riferimenti si modificano (verificato con hashCode)
49     prestiti.ricostruisciPrestiti(risorse);
50
51     Storico storico = new Storico(prestiti, fruitori, risorse);
52
53     // coordina i compiti
54     MainFacade mainFacade = new MainFacade(risorse, fruitori, prestiti, storico, gestoreSalvataggi);
55     mainFacade.start();
56 }
57 }
```

Open-Closed Principle (OCP)

Classi aperte all'estensione, chiuse alle modifiche

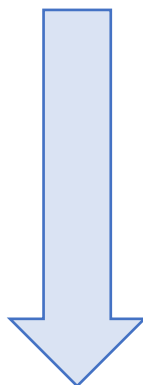
Abbiamo considerato come principale possibilità di cambiamento dei requisiti del progetto l'aggiunta di una nuova tipologia di Risorsa: per questo motivo abbiamo utilizzato in tutto il codice l'interfaccia *Risorsa*, in modo da non dover poi andare a cercare in tutte le classi i riferimenti alle risorse 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.

```
Prestiti.java  Main.java x  ManagePrestitiHandler.java  PrestitiController.java
821 case 4: //RICHIEDI PRESTITO (non in prestiti perchè devo poter accedere alle risorse)
822 {
823     MyMenu menu = new MyMenu("scegli la categoria di risorsa: ", CATEGORIE);
824
825     try
826     {
827         String categoria = CATEGORIE[menu.scegliBase() - 1]; //stampa il menu (partendo da 1 e non da 0) con i generi e ritorna quello selezionato
828         if(categoria == CATEGORIE[0])// == "Libri"
829         {
830             if(prestiti.numPrestitiAttiviDi(utenteLoggato, categoria) == Libro.PRESTITI_MAX)
831             {
832                 System.out.println("\nNon puoi prenotare altri " + categoria + ": "
833                     + "\nHai raggiunto il numero massimo di risorse in prestito per questa categoria");
834             }
835             else//può chiedere un altro prestito
836             {
837                 Libro libro = archivio.getLibri().scegliLibro();
838
839                 if(libro != null)
840                 {
841                     if(prestiti.prestitoFattibile(utenteLoggato, libro))
842                     {
843                         prestiti.addPrestito(utenteLoggato, libro);
844
845                         System.out.println(libro.getTitolo() + " prenotato con successo!");
846                     }
847                     else//!prestitoFattibile se l'utente ha già una copia in prestito
848                     {
849                         System.out.println("Prenotazione rifiutata: possiedi già questa risorsa in prestito");
850                     }
851                 }
852                 // qui libro==null: vuol dire che l'utente non ha selezionato un libro (0: torna indietro)
853             }
854         }
855         else if(categoria == CATEGORIE[1])// == "Films"
856         {
857             if(prestiti.numPrestitiAttiviDi(utenteLoggato, categoria) == Film.PRESTITI_MAX)
858             {
859                 System.out.println("\nNon puoi prenotare altri " + categoria + ": "
860                     + "\nHai raggiunto il numero massimo di risorse in prestito per questa categoria");
```

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)

```
Prestiti.java  Main.java  ManagePrestitiHandler.java  PrestitiController.java
26
27 public void richiediPrestito(int scelta)
28 {
29     try
30     {
31         String categoria = CATEGORIE[scelta - 1];
32
33         Risorsa risorsa = risorseController.scegliRisorsa(categoria);
34         if(risorsa != null)
35         {
36             prestitiController.effettuaPrestito(utenteLoggato, risorsa);
37         }
38
39         gestoreSalvataggi.salvaPrestiti();
40         gestoreSalvataggi.salvaRisorse();
41     }
42     catch(ArrayIndexOutOfBoundsException e)
43     {
44         // se utente seleziona 0 (INDIETRO) -> CATEGORIE[-1] dà eccezione
45         // corrisponde ad ANNULLA, non va fatto nulla
46     }
47 }
48
```

```
Prestiti.java  Main.java  ManagePrestitiHandler.java  PrestitiController.java
64 //addPrestito può lanciare due eccezioni diverse: o il fruitore possiede già la risorsa o ha raggiunto
65 public void effettuaPrestito(Fruitore fruitore, Risorsa risorsa)
66 {
67     try
68     {
69         model.addPrestito(fruitore, risorsa);
70         prestitiView.prenotazioneEffettuata(risorsa);
71     }
72     catch(RaggiunteRisorseMaxException e)
73     {
74         prestitiView.raggiunteRisorseMassime(risorsa.getSottoCategoria());
75     }
76     catch(RisorsaGiàPossedutaException e1)
77     {
78         prestitiView.risorsaPosseduta();
79     }
80 }
```


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.

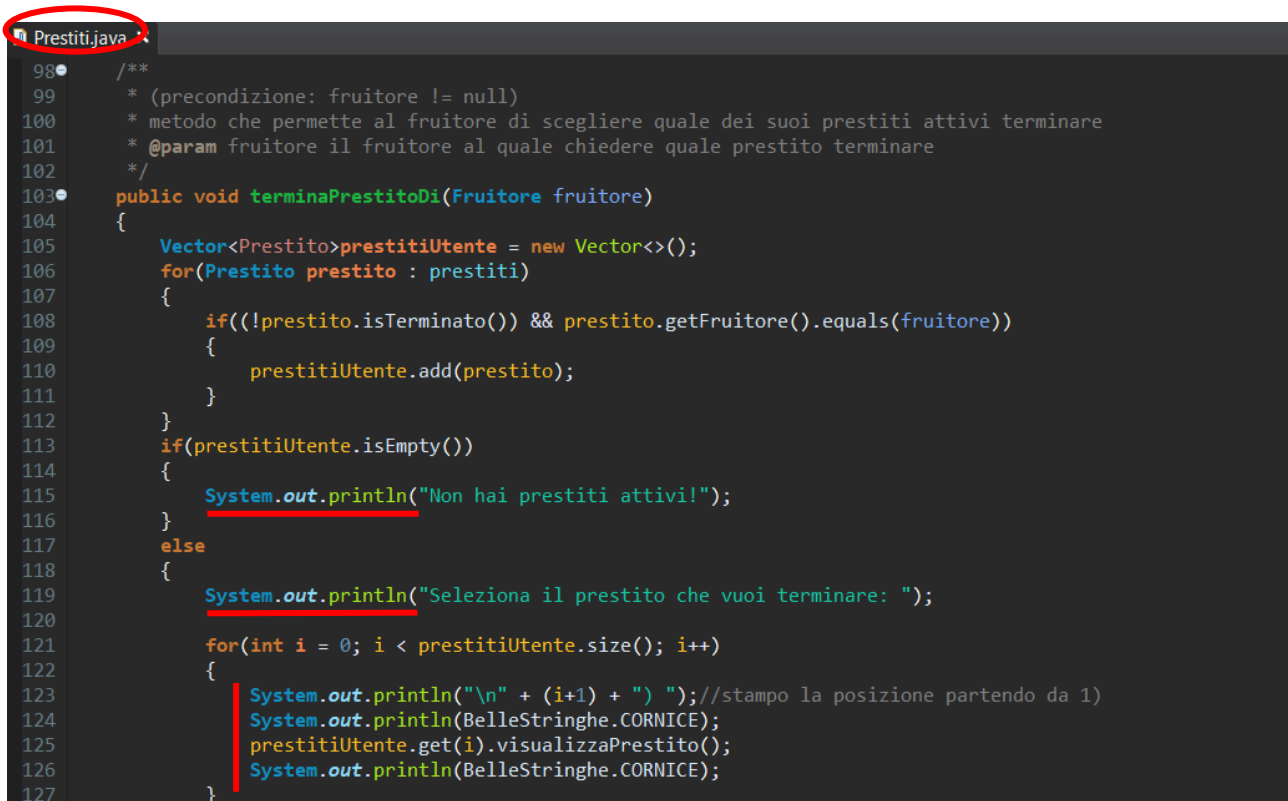
Infine abbiamo messo la loro istanziazione all'interno dei Controller MVC, facendo riferimento solo alle interfacce.

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:

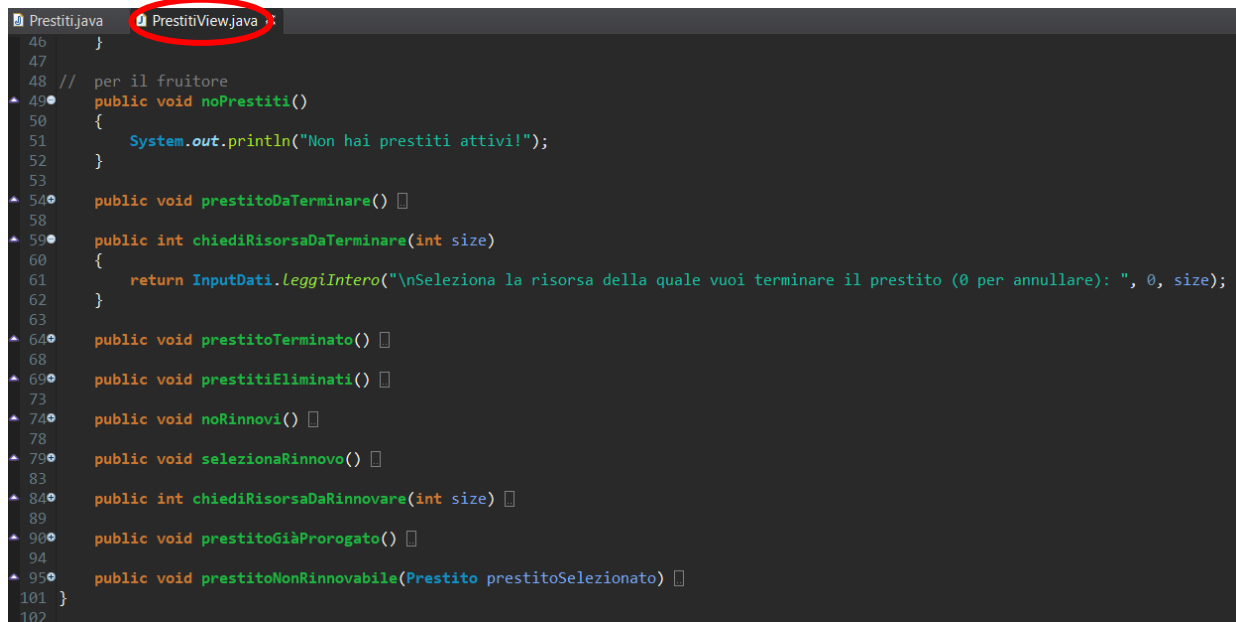
metodi per stampare a console "immersi" all'interno del codice



```
98  /**
99  * (precondizione: fruitore != null)
100  * metodo che permette al fruitore di scegliere quale dei suoi prestiti attivi terminare
101  * @param fruitore il fruitore al quale chiedere quale prestito terminare
102  */
103  public void terminaPrestitoDi(Fruitore fruitore)
104  {
105      Vector<Prestito>prestitiUtente = new Vector<>();
106      for(Prestito prestito : prestiti)
107      {
108          if(!prestito.isTerminato()) && prestito.getFruitore().equals(fruitore))
109          {
110              prestitiUtente.add(prestito);
111          }
112      }
113      if(prestitiUtente.isEmpty())
114      {
115          System.out.println("Non hai prestiti attivi!");
116      }
117      else
118      {
119          System.out.println("Seleziona il prestito che vuoi terminare: ");
120
121          for(int i = 0; i < prestitiUtente.size(); i++)
122          {
123              System.out.println("\n" + (i+1) + " "); //stampo la posizione partendo da 1)
124              System.out.println(BelleStringhe.CORNICE);
125              prestitiUtente.get(i).visualizzaPrestito();
126              System.out.println(BelleStringhe.CORNICE);
127          }
128      }
129  }
```

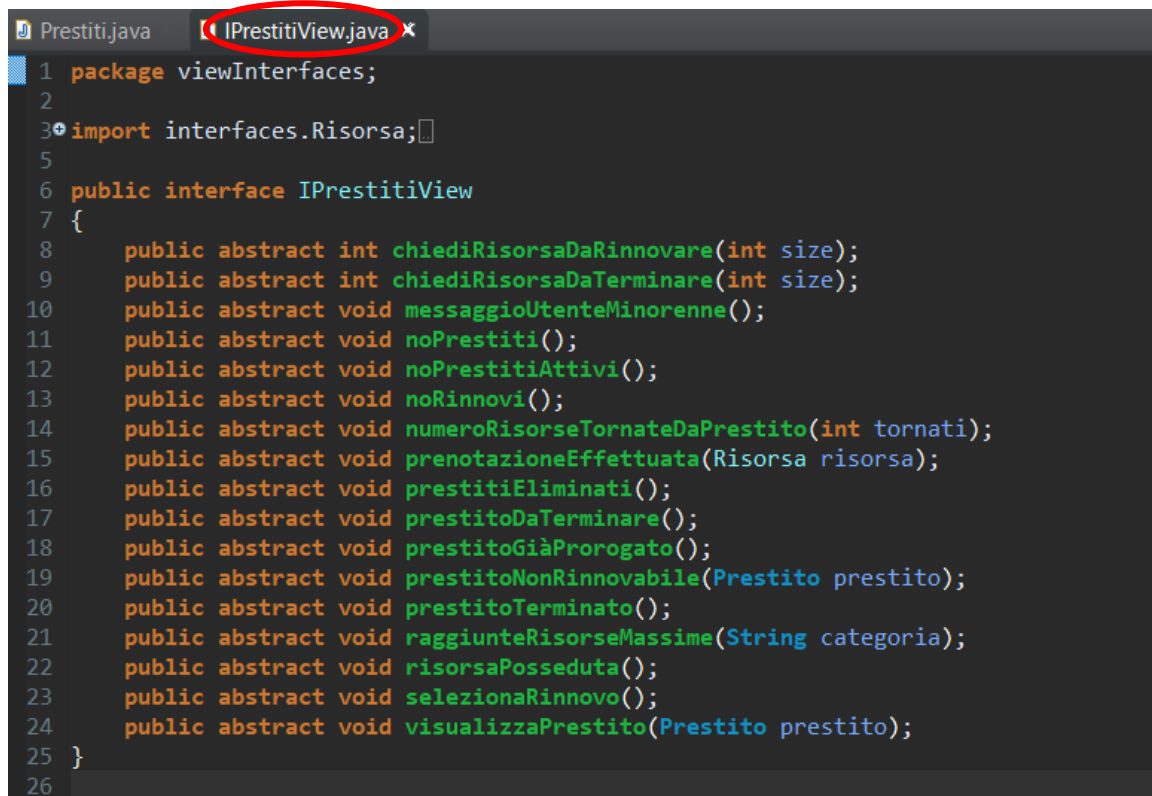
Progetto nuovo:

Classe View



```
46 }
47
48 // per il fruitore
49 public void noPrestiti()
50 {
51     System.out.println("Non hai prestiti attivi!");
52 }
53
54 public void prestitoDaTerminare() {}
55
56 public int chiediRisorsaDaTerminare(int size)
57 {
58     return InputDati.leggiIntero("\nSeleziona la risorsa della quale vuoi terminare il prestito (0 per annullare): ", 0, size);
59 }
60
61 public void prestitoTerminato() {}
62
63 public void prestitiEliminati() {}
64
65 public void noRinnovi() {}
66
67 public void selezionaRinnovo() {}
68
69 public int chiediRisorsaDaRinnovare(int size) {}
70
71 public void prestitoGiàProrogato() {}
72
73 public void prestitoNonRinnovabile(Prestito prestitoSelezionato) {}
74
75 }
```

Interface IView



```
1 package viewInterfaces;
2
3 import interfaces.Risorsa;
4
5
6 public interface IPrestitiView
7 {
8     public abstract int chiediRisorsaDaRinnovare(int size);
9     public abstract int chiediRisorsaDaTerminare(int size);
10    public abstract void messaggioUtenteMinorenne();
11    public abstract void noPrestiti();
12    public abstract void noPrestitiAttivi();
13    public abstract void noRinnovi();
14    public abstract void numeroRisorseTornateDaPrestito(int tornati);
15    public abstract void prenotazioneEffettuata(Risorsa risorsa);
16    public abstract void prestitiEliminati();
17    public abstract void prestitoDaTerminare();
18    public abstract void prestitoGiàProrogato();
19    public abstract void prestitoNonRinnovabile(Prestito prestito);
20    public abstract void prestitoTerminato();
21    public abstract void raggiunteRisorseMassime(String categoria);
22    public abstract void risorsaPosseduta();
23    public abstract void selezionaRinnovo();
24    public abstract void visualizzaPrestito(Prestito prestito);
25 }
26
```

(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 tramite 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.

```
13
14 public class PrestitiController
15 {
16     private Prestiti model;
17     private IPrestitiView prestitiView;
18     private IMessaggiSistemaView messaggiSistemaView;
19
20     public PrestitiController(Prestiti prestiti)
21     {
22         model = prestiti;
23         // per prestitiView: sennò Controller dipenderebbe da PrestitiView, a causa dell'istanziamiento. così solo Interface
24         // stessa cosa per messaggi sistema view
25         try
26         {
27             this.prestitiView = (IPrestitiView)Class.forName(System.getProperty("PrestitiView")).newInstance();
28             this.messaggiSistemaView = (IMessaggiSistemaView)Class.forName(System.getProperty("MessaggiSistemaView")).newInstance();
29         }
30         catch (InstantiationException | IllegalAccessException | ClassNotFoundException e)
31         {
32             e.printStackTrace();
33         }
34     }
35 }
```

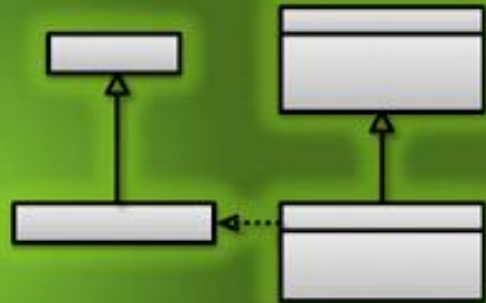
File Config.properties:

```
config.properties
1  FruitoriView = view.FruitoriView
2  FilmsView = view.FilmsView
3  LibriView = view.LibriView
4  MessaggiSistemaView = view.MessaggiSistemaView
5  PrestitiView = view.PrestitiView
6  StoricoView = view.StoricoView
7  SavesManager = service.GestoreSalvataggi
8
```

4)

Gang of Four (GoF) Patterns

```
class DocumentFactory{  
public:  
virtual Document * Create() = 0 ;           //Factory method  
Document * CreateDocument(){               //An operation  
    //Preprocessing  
    //Check for permissions, locks, etc  
    Document *d = Create() ;  
    //Postprocessing  
    //Verify integrity, apply some common operation, etc  
    return d ;  
}  
virtual ~DocumentFactory(void){  
}  
};
```

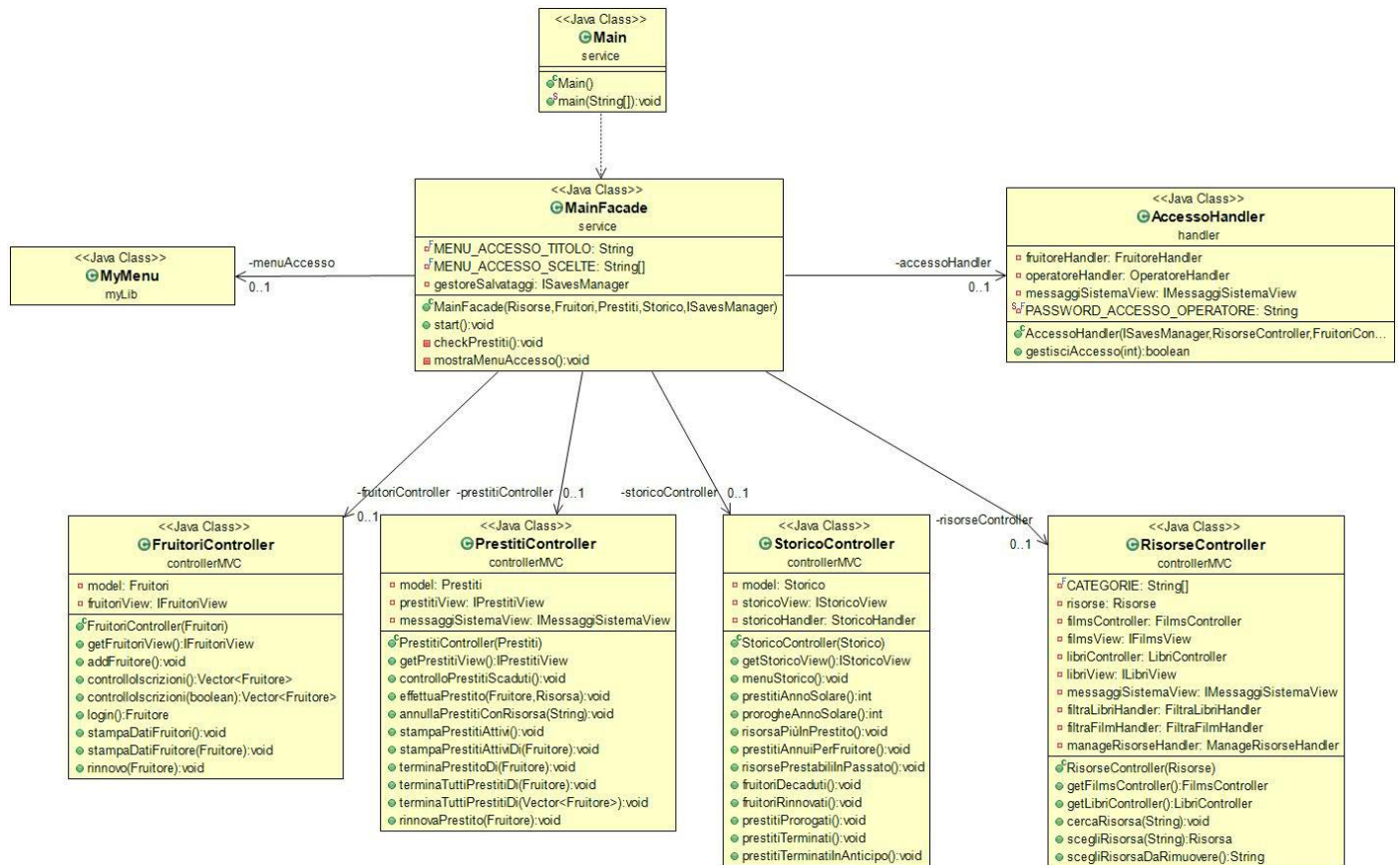


Gang of Four Design Patterns

Facade

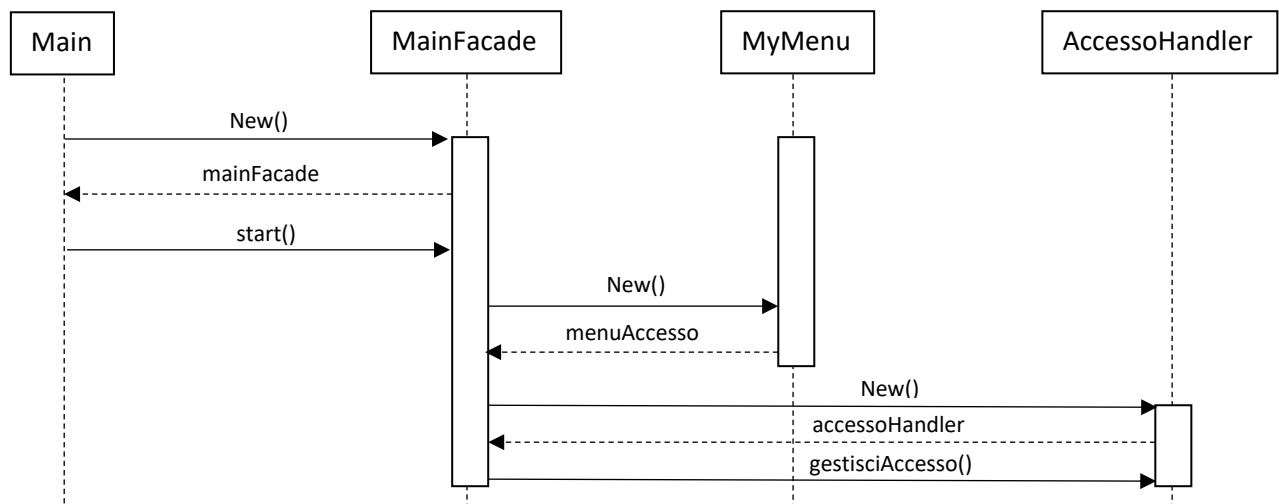
Il Main non deve interagire con tutte le componenti del sottosistema. Delega la gestione delle funzionalità del sottosistema definendo un punto di contatto con esso, nel nostro caso la classe MainFacade: tale classe si occupa di coordinare la collaborazione tra i vari componenti del sistema, in particolare 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.

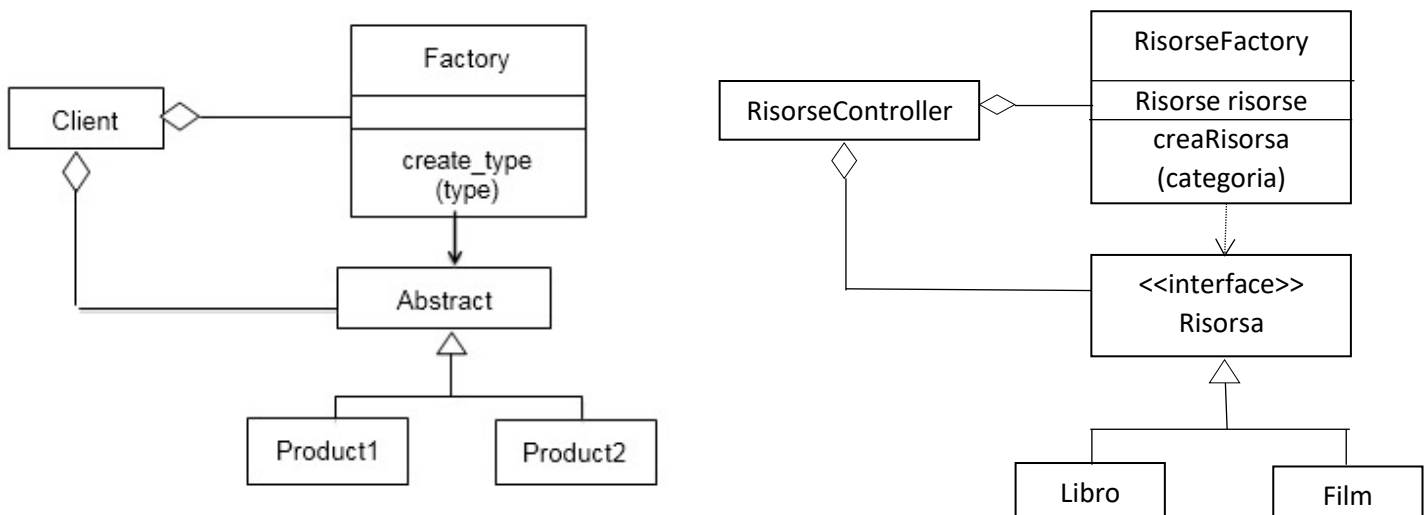


Il Main ha una dipendenza dal MainFacade, che crea i Controller, avvia il menu iniziale (di accesso) e delega a sua volta i compiti successivi ad AccessoHandler, che in base alla scelta dell’utente nel menu deciderà cosa fare.

Diagramma di sequenza (semplificato):



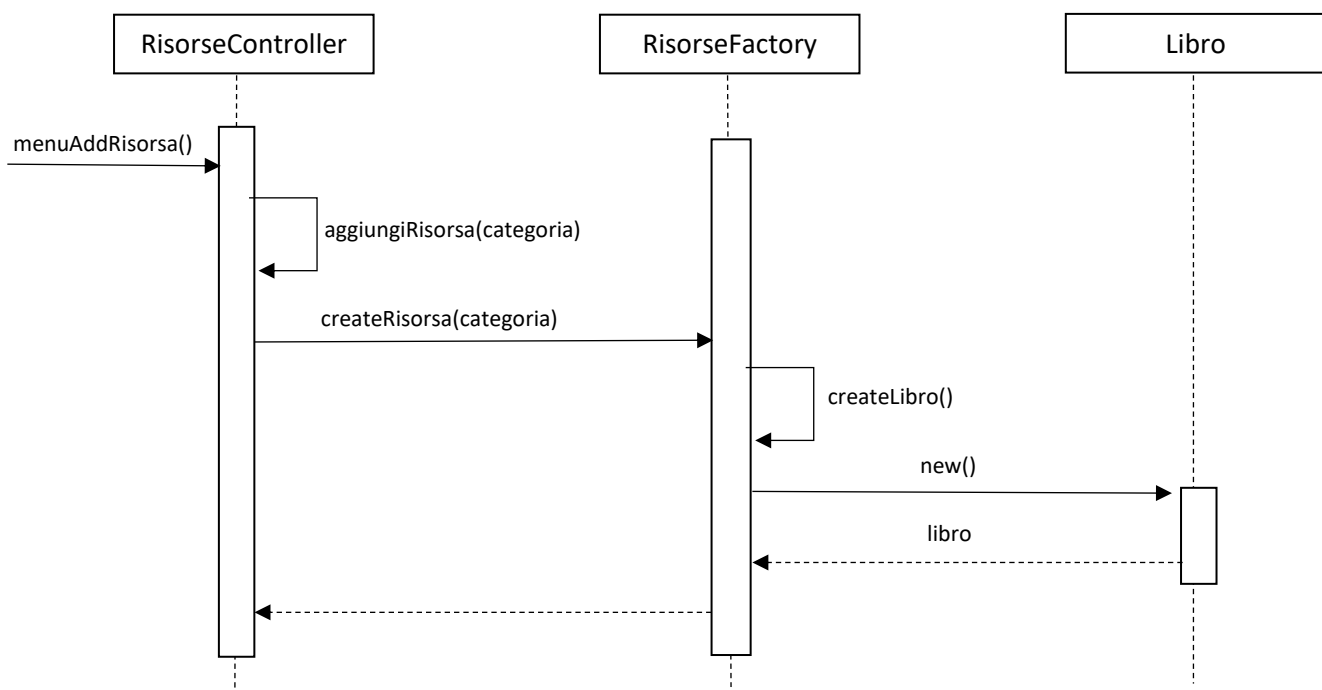
Factory



- **Client** -> controllerMVC.RisorsController
- **Factory** -> service.RisorsFactory
- **Abstract** -> interfaces.Risorsa
- **Product1** -> model.Libro
- **Product2** -> model.Film

RisorsFactory crea l'oggetto richiesto da *RisorsController* (il client):

Risorsa è l'interfaccia che implementano tutti gli oggetti che possono essere creati (nel nostro caso: *Libro* e *Film*).

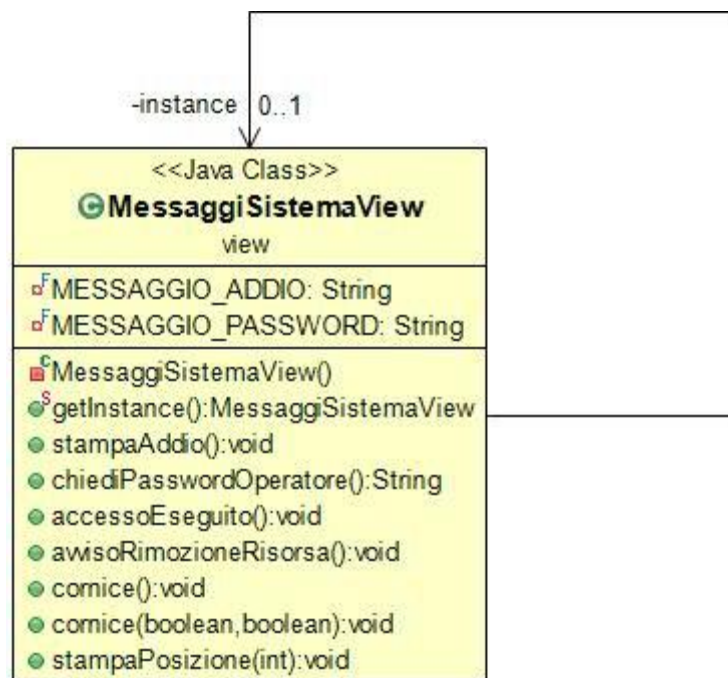


Singleton

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()`).

```
7 public class MessaggiSistemaView implements IMessaggiSistemaView
8 {
9 // SINGLETON: serve in molti punti e non voglio nè creare oggetto nuovo ogni volta nè farlo passare da un'oggetto all'altro
10 private static MessaggiSistemaView instance;
11 // costruttore privato, usato solo da getInstance()
12 private MessaggiSistemaView() {}
13 // se MessaggiSistemaView è già stata istanziata viene restituita l'istanza, altrimenti viene creata
14 public static MessaggiSistemaView getInstance()
15 {
16     if(instance == null)
17     {
18         instance = new MessaggiSistemaView();
19     }
20     return instance;
21 }
22 }
```



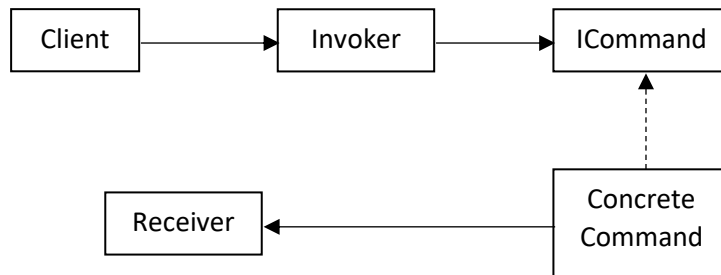
Facendo ciò è però sorto un problema, legato al punto precedente, cioè l’istanziare gli oggetti View usando proprietà di sistema: il metodo `newInstance()` invoca infatti il costruttore della classe, che nel caso del Singleton non andrebbe usato. Abbiamo quindi cambiato strategia di istanziazione, per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

In particolare, in `PrestitiController` e `RisorseController`:

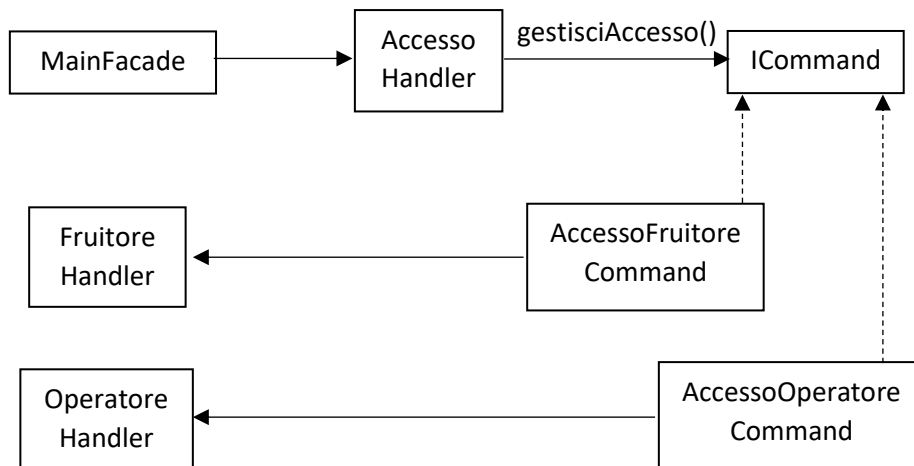
```
30 MessaggiSistemaView è un SINGLETON, quindi il costruttore è privato e facendo "Class.forName(...).getInstance()" il costruttore non può venire
31 invocato: Allora una volta ottenuta la classe con "Class.forName(...)" richiamiamo il metodo statico "getInstance()" tipico dei singleton.
32 essendo il metodo statico i parametri dei metodi non servono e possono essere null
33 this.messaggiSistemaView = (IMessaggiSistemaView)Class
34     .forName(System.getProperty("MessaggiSistemaView"))
35     .getMethod("getInstance", (Class<?>[])null)
36     .invoke(null, (Object[])null);
```

Command

Pattern:



Implementazione:



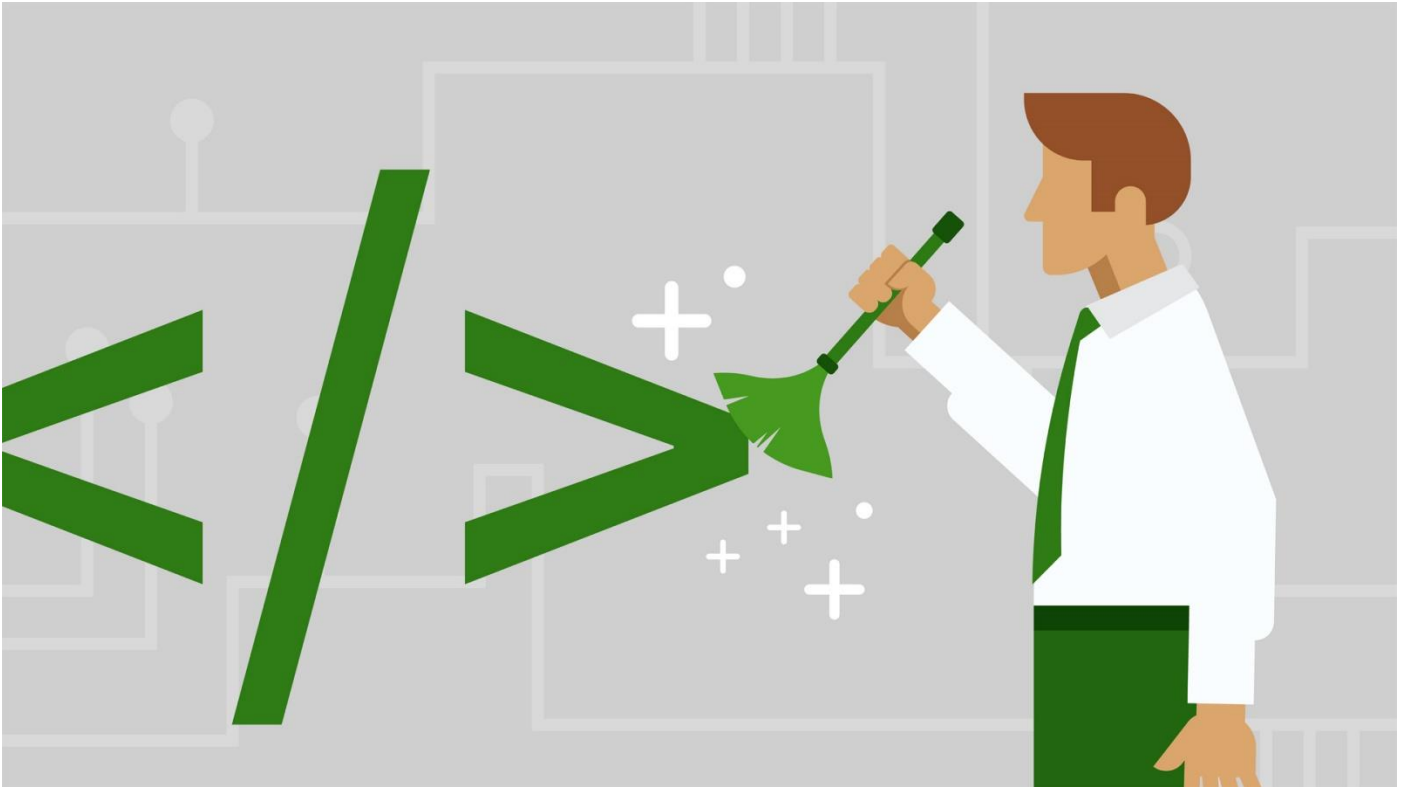
- Client: MainFacade
- Invoker: AccessoHandler
- Receiver: FruitoreHandler/OperatoreHandler

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.

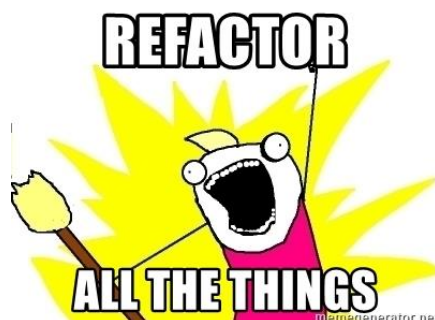
5) REFACTORING



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. Tali test sono automatizzati (JUnit) in modo che essi stessi “verifichino i propri risultati” (lasciandoci solo il compito di controllare il colore di tale verifica: se verde -> ok)

(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 la suddivisioni delle classi, prima poche e dense di responsabilità diverse, per rispettare il modello MVC, separando quindi la logica applicativa, la parte di presentazione e l’interazione con l’utente.



Per sistemare metodi troppo lunghi:

1) EXTRACT METHOD

Serve per dividere metodi troppo lunghi in metodi più brevi e semplici da comprendere.

Serve per evitare commenti al codice (che sono spesso “deodorante” per la puzza del codice), mettendo parte del codice dentro un metodo il cui nome ne spieghi lo scopo.

- Parte5/Films.scegliFilm()

Metodo molto lungo (righe 287-414, evitiamo il copia-incolla qui), con all’interno la creazione e gestione del menu, compiti di presentazione, tutta la logica di filtro per i film e la visualizzazione dell’archivio completo.

REFACTORING



- Refactoring/handler/ManageRisorseHandler.scegliFilm()

```
public Risorsa scegliFilm(int scelta)
{
    switch(scelta)
    {
        case 0://INDIETRO
        {
            return null;
        }
        case 1://FILTRA RICERCA
        {
            Vector<Risorsa>filmsFiltrati = filtraFilmHandler.menuFiltraFilm(true);
            return risorseController.selezionaRisorsa(filmsFiltrati, "film");
        }
        case 2://VISUALIZZA ARCHIVIO
        {
            Vector<Risorsa>filmPrestabili = risorseController.getFilmsController().filmsPrestabili();
            return risorseController.selezionaRisorsa(filmPrestabili, "film");
        }
    }
    // DEFAULT: qua non arriva mai
    assert false;
    return null;
}
```

Questo metodo è un handler in quanto riceve la scelta ottenuta dall’interazione dell’utente con la UI (menu creato in FilmsController.menuScegliFilm()) e ne gestisce la logica, delegando i compiti.

I vari compiti che erano in precedenza all’interno del metodo sono stati gestiti attraverso metodi più semplici il cui nome spiega la loro funzione (filtraFilm, filmsPrestabili, selezionaRisorsa).

2) MOVE METHOD

(quando un metodo usa e/o è usato da più metodi di un'altra classe)

Sempre facendo riferimento all'esempio precedente, i "metodi estratti" sono stati spostati nelle classi più consone, come per esempio *RisorseController*, che possiede tutte le informazioni necessarie per la selezione di una risorsa (l'elenco delle risorse stesse e la capacità di distinguere tra risorse prestabili e non)

3) EXTRACT CLASS

Per quanto riguarda invece le funzionalità di filtro (per titolo, per anno, per regista, ...), esse sono state raccolte in una classe apposita, che abbiamo chiamato FiltraFilmHandler, in modo che le singole classi gestiscano poche e chiare responsabilità.

```
1 package handler;
2
3 import java.util.Vector;
4
5
6
7
8 public class FiltraFilmHandler
9 {
10     private FilmsController filmsController;
11
12     public FiltraFilmHandler(FilmsController filmsController)
13     {
14         this.filmsController = filmsController;
15     }
16
17     /**
18      * se i film vengono filtrati per essere prenotati viene restituita la lista, se invece è solo per la ricerca vengono visualizzati e basta
19      * @param daPrenotare
20      * @return la lista dei film filtrati (if daPrenotare==true)
21      */
22     public Vector<Risorsa> menuFiltraFilm(boolean daPrenotare) {}
23
24     public Vector<Risorsa> filtraFilm(int scelta, boolean daPrenotare) {}
25
26     public Vector<Risorsa> filtraFilmPerTitolo(String titoloParziale){}
27
28     public Vector<Risorsa> filtraFilmPerUscita(int annoUscita){}
29
30     public Vector<Risorsa> filtraFilmPerRegista(String regista){}
```

Per semplificare statements condizionali:

4) INTRODUCE ASSERTIONS

Quando il codice fa assunzioni -> rendile esplicite tramite asserzioni.

Dato che durante lo svolgimento del progetto abbiamo incontrato vari problema riguardanti il salvataggio/caricamento da file, abbiamo inserito delle assertions subito dopo i caricamenti, all'interno del Main, in modo da sapere fin dall'inizio se gli oggetti fossero stati effettivamente caricati (risparmiando tempo e successive NullPointerException).

```
*Main.java x
45     Fruitori fruitori = gestoreSalvataggi.getFruitori();
46     assert fruitori != null;
47     Risorse risorse = gestoreSalvataggi.getRisorse();
48     assert risorse != null;
49     Prestiti prestiti = gestoreSalvataggi.getPrestiti();
50     assert prestiti != null;
51
```

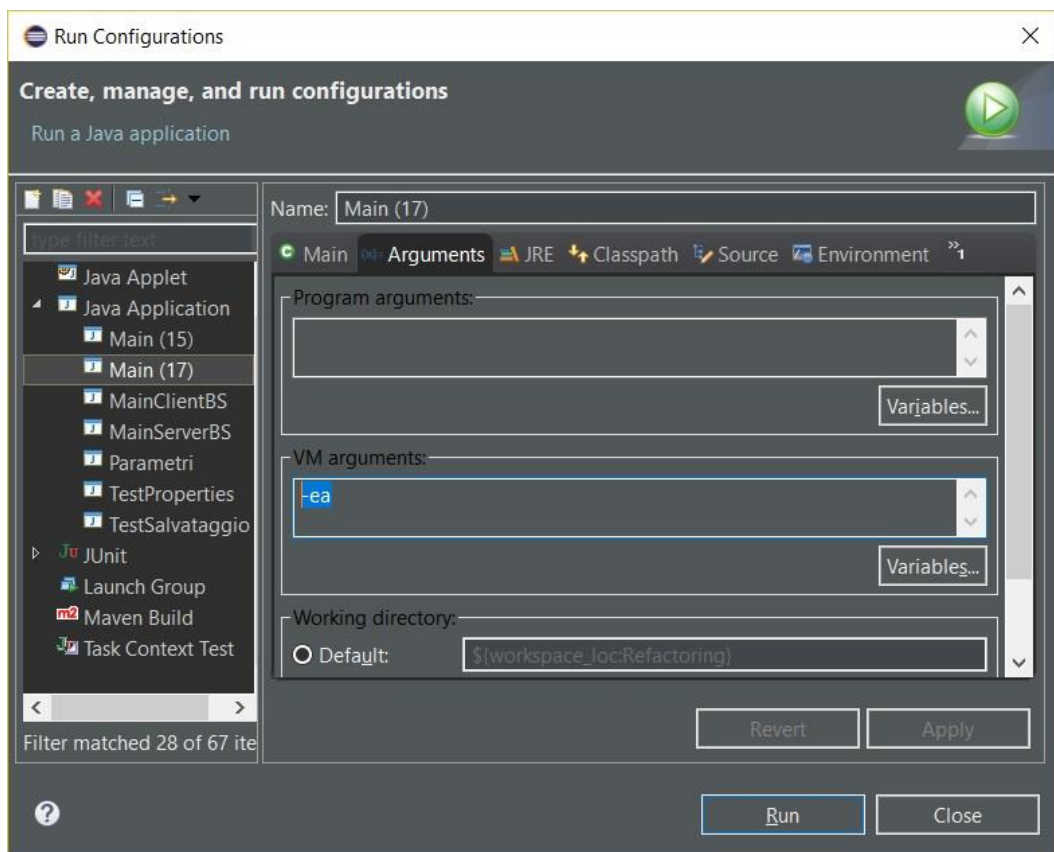
Abbiamo usato le *assertions* anche nelle parti di codice (principalmente nei menu) dove l'esecuzione non deve mai arrivare: la logica era già gestita correttamente ma l'abbiamo ritenuta una sicurezza in più.

Un esempio è il metodo *scegliFilm()*, il cui codice è già riportato in: [1\) EXTRACT METHOD](#):

dove l'esecuzione non deve arrivare abbiamo inserito un "*assert false*" (che da sempre errore), in modo da non ritrovarci con valori nulli senza saperlo.

Ovviamente queste asserzioni non vanno gestite con try/catch, perderebbero la loro utilità!

(Le assertions vanno abilitate all'interno del proprio IDE: su Eclipse siamo andati in Run -> Run configurations -> Arguments, e abbiamo inserito "-ea" (enable assertions) all'interno di "VM Arguments", in modo da non doverle abilitare manualmente ogni volta.)



Per rendere le chiamate ai metodi più semplici

5) REPLACE CONSTRUCTOR WITH FACTORY METHOD

Abbiamo sostituito la chiamata diretta ai costruttori di *Libro* e *Film* (che avveniva nelle classi *Libri* e *Films*, che ora non esistono più), utilizzando il pattern *Factory* (GoF): in questo modo *RisorseController* può delegare tutta la logica di creazione delle risorse alla classe *RisorseFactory* (Refactoring/service/*RisorseFactory*), specificando solamente il tipo di risorsa desiderata. Questo è possibile grazie all'uso dell'interface *Risorsa*.

PRIMA:

(Parte5/Libri.addLibro())

```
public void addLibro()
{
    String sottoCategoria = scegliSottoCategoria();//la sottocategoria della categoria LIBRO (Romanzo, fumetto, poesia,...)
    // se l'utente annulla la procedura
    if(sottoCategoria == "annulla")
    {
        return;
    }
    String titolo = InputDati.leggiStringaNonVuota("Inserisci il titolo del libro: ");
    int pagine = InputDati.leggiInteroPositivo("Inserisci il numero di pagine: ");
    int annoPubblicazione = InputDati.leggiInteroConMassimo("Inserisci l'anno di pubblicazione: ", GestioneDate.ANNO_CORRENTE);
    String casaEditrice = InputDati.leggiStringaNonVuota("Inserisci la casa editrice: ");
    String lingua = InputDati.leggiStringaNonVuota("Inserisci la lingua del testo: ");
    Vector<String> autori = new Vector<String>();
    do
    {
        String autore = InputDati.leggiStringaNonVuota("Inserisci l'autore: ");
        autori.add(autore);
    }
    while(InputDati.yesOrNo("ci sono altri autori? "));
    String genere = this.scegliGenere(sottoCategoria);//se la sottocategoria ha generi disponibili
    int nLicenze = InputDati.leggiInteroPositivo("Inserisci il numero di licenze disponibili: ");

    Libro l = new Libro("L"+lastId++, sottoCategoria, titolo, autori, pagine, annoPubblicazione, casaEditrice, lingua, genere, nLicenze);

    if(!libroEsistente(l))
    {
        addPerSottoCategorie(l);
        System.out.println("Libro aggiunto con successo!");
    }
    else
    {
        System.out.println("Il libro è già presente in archivio");
    }
}
```

(Parte5/Films.addFilm())

```
public void addFilm()
{
    String sottoCategoria = this.scegliSottoCategoria();//la sottocategoria della categoria FILM ("Azione","Avventura","Fantascienza...")
    // se l'utente annulla la procedura
    if(sottoCategoria == "annulla")
    {
        return;
    }
    String titolo = InputDati.leggiStringaNonVuota("Inserisci il titolo del film: ");
    int durata = InputDati.leggiInteroPositivo("Inserisci la durata del film (in minuti): ");
    int annoDiUscita = InputDati.leggiIntero("Inserisci l'anno di uscita: ", ANNO_PRIMA_PELLICOLA, GestioneDate.ANNO_CORRENTE);
    String lingua = InputDati.leggiStringaNonVuota("Inserisci la lingua del film: ");
    String regista = InputDati.leggiStringaNonVuota("Inserisci il regista: ");
    int nLicenze = InputDati.leggiInteroPositivo("Inserisci il numero di licenze disponibili: ");

    Film f = new Film("F"+lastId++, sottoCategoria, titolo, regista, durata, annoDiUscita, lingua, nLicenze);
    addPerSottoCategorie(f);

    System.out.println("Film aggiunto con successo!");
}
```

DOPO:

(Refactoring/controllerMVC/RisorseController.aggiungiRisorsa())

```
// utilizza FACTORY
public void aggiungiRisorsa(String categoria)
{
    if(risorseFactory == (null))
    {
        risorseFactory = new RisorseFactory(risorse, libriController.getLibriView(), filmsController.getFilmsView());
    }
    ➔ Risorsa risorsa = risorseFactory.creaRisorsa(categoria);

    // utente ha annullato
    if(risorsa == (null))
    {
        return;
    }

    boolean aggiuntaRiuscita = risorse.addRisorsa(risorsa);
    if(aggiuntaRiuscita)
    {
        risorseView.aggiuntaRiuscita(risorsa.getClass());
    }
    else
    {
        risorseView.aggiuntaNonRiuscita(risorsa.getClass());
    }
}
```

The End