UNIVERSITÁ DEGLI STUDI DI SALERNO

Ingegneria del Software

OBJECT DESIGN DOCUMENT

ANNO ACCADEMICO 2018/2019

a

TOP MANAGER:

Nome
Prof. Andrea De Lucia

PARTECIPANTI:

Nome	Matricola
Mario Sessa	0512104650
Luigi Crisci	0512104740
Pasquale Ambrosio	0512104704

HISTORY:

Data	Versione	Cambiamenti	Autori
26/12/2018	1.0	Creazione capitolo 1.	Mario Sessa
27/12/2018	1.1	Creazione capitolo 2.	Mario Sessa
28/12/2018	1.2	Completamento capitolo 3 e 4.	Mario Sessa
14/01/2019	2.0	Modifica capitolo 3	Luigi Crisci
18/01/2019	2.1	Inserimento parziale dei contratti	Mario Sessa
14/02/2019	2.2	Completamento ODD e supervisione del documento	Mario Sessa

Indice

1. INTRODUZIONE

1.1 Object Design Trade-offs

Dopo la realizzazione dei documenti RAD e SDD abbiamo descritto in linea di massima quello che sarà il nostro sistema e gli obbiettivi da seguire, tralasciando gli aspetti implementativi. Il seguente documento ha lo scopo di produrre un modello capace di integrare in modo coerente e preciso tutte le funzionalità individuate nelle fasi precedenti. In particolare, definisce le interfacce delle classi, le operazioni, i tipi, gli argomenti e le signature dei sottosistemi definiti nel System Design. Inoltre, sono specificati i trade-off e le linee guida.

Comprensibilità vs Tempo

Il codice deve essere il più chiaro possibile, ogni componente deve essere accompagnato da un commento in grado di descrivere quali operazioni si stanno implementando. Questa forma di comprensibilità del codice porterà dei rallentamenti in fase implementativa e di testing andando a creare, però, vantaggi sulla comprensione globale del sistema e delle sue componenti.

Prestazioni vs Costi

Non avendo finanziamenti esterni, si utilizzeranno delle tecnologie open-source in grado di gestire il sistema in maniera gratuita. Nello specifico, verrà utilizzato un database relazionale come repository centrale per i dati gestiti dal sistema e un web server monolitico per la gestione dell'interazione con gli utenti.

Interfaccia vs Usabilità

L'interfaccia verrà gestita in modo tale da poter essere il più semplice ed intuitiva possibile, attraverso l'uso di form e bottoni di facile comprensione per l'utente finale.

Sicurezza vs Efficienza

Il sistema si baserà prevalentemente sulla gestione della sicurezza per evitare accessi non autorizzati cosi da proteggere informazioni personali quali e-mail, password e carte di credito.

1.2 Linee guida per la documentazione delle interfacce

Gli sviluppatori dovranno seguire le corrispondenti linee guida durante la fase implementativa del progetto:

1.2.1 Naming Convention

I nomi utilizzati per la rappresentazione dei concetti principali, delle funzionalità e delle componenti generiche del sisterma devono rispettare le seguenti condizioni:

1. I **nomi** devono essere:

a. Appartenenti alla lingua italiana, se possibile.
b. Di lunghezza medio-breve
c. Non sostituiti da acronimi o abbreviazioni di alcun genere
d. Composti da caratteri compresi in [0-9, a-z, A-Z]

2. Le **variabili** devono:
   a. Rispettare la Camel Notation
   b. Iniziare con lettere minuscole
   c. Essere composti da caratteri compresi in [0-9, a-z,A-Z]

È possibile far iniziare le variabili statiche con "_" cosi da contraddistinguerle dal resto delle variabili presenti all'interno del codice. Questa distinzione è utile per evitare errori sull'uso improprio di tali forme di variabili.

3. Le **classi e le interfacce** devono:
   a. Rispettare la Camel Notation
   b. Iniziare con la lettera grande
   c. Concludersi con il tipo di elemento che rappresentano:
      Esempio:
      ```
      public class UtenteBean {} /* Per una classe di tipo Bean */
      public class TestClass {} /* Per una classe generica */
      public interface DataInterface /* Per un'interfaccia */
      ```

4. Le variabili **costanti** devono:
   a. Utilizzare solamente caratteri maiuscoli
   b. Separare i vari nomi che la compongono da "_"
   c. Evitare di iniziare con "_"
   d. Contenere solamente caratteri [a-z, A-Z, 0-9]
   e. Essere di lunghezza medio-breve

5. I **pacchetti** devono:
   a. Contenere solamente caratteri minuscoli
   b. Contenere solamente caratteri a-z
   c. Di semantica affine con gli elementi di cui è composto

6. I **metodi** devono:
   a. Iniziare con lettere minuscole
   b. Rispettare la Camel Notation
   c. Evitare di iniziare con GET o SET se non si trattano di metodi setting o getting della classe corrispondente.
   d. Contenere solamente caratteri [a-z, A-Z]

7. Le pagine **JSP**:
   a. Contengono solamente caratteri minuscoli
   b. Sono di lunghezza medio-breve
   c. Hanno nomi composte da una singola parola in inglese o in italiano

1.2.2 Gestione formattazione del codice

Per rispettare i criteri di comprensibilità definiti nella sottosezione precedente, bisogna approcciarsi con particolare cura sull'indentazione del codice e su come alcuni elementi del codice devono essere formattati. Di seguito verranno descritti alcuni esempi per dare una visione generale al programmatore su come si vuole indentare il codice.

1.2.2.1 Gestione codice HTML e XML

```
<html>

        <head>

        <head>

    <body>

                <div>

                    Contenuto DIV

            <div>

        <body>

<html>
```

Il codice HTML deve essere indentato in maniera tale da poter mantenere sulla stessa colonna il TAG di apertura e di chiusura. Inoltre, il contenuto di un TAG si distanzierà dalle clausole dell'elemento in cui è contenuto di una distanza pari ad 1 TAB.

I Tag che non hanno una clausola di chiusura seguiranno solamente la seconda condizione.

1.2.2.2 Gestione codice classi Java e Servlet.

All'interno di questa sezione ci soffermeremo sulla corretta formattazione delle classi Java, delle Servlet e dell'uso di codice Javadoc corrispondente.

Le classi **Java** devono seguire le seguenti condizioni:

1. Il codice Javadoc deve essere utilizzato per la descrizione di:
    1. Classi
    2. Metodi
    3. Interfacce

2. I metodi, le classi interne e le variabili devono essere indentati in colonne successive a quella della classe o del metodo in cui sono contenute.

Un **esempio** più pratico del corretto formato lo vediamo nel frammento di codice seguente:

```
package com;

/**
 *
 * @author Mario Sesssa
 * @version 1.1
 * @since 26/12/2018
 *
 */

public class example {

    private int x;

    /**
     *  Costruttore della classe
     * @param x
     *
     */

    public example(int x) {
        this.x = x;
    }

    /**
     *   Effettua la somma tra la variabile di istanza e il parametro passato.
     *
     * @param int
     * @return int
     */
    public int sum(int y) {
        return this.x = this.x + y;
    }


}
```

Il codice Javadoc di ogni classe deve contenere le clausole @author, @versione e @since. Mentre ogni metodo deve contenere obbligatoriamente una descrizione delle operazioni o della funzionalità che esegue e può richiamare clausole come @param e @return.

Il formato delle classi Servlet, invece, deve:

1. Contenere un costruttore, anche se vuoto
2. Contenere i metodi doGet() e doPost()
3. Contenere il codice Javadoc per la classe e per i metodi doGet() e doPost() in modo da definire lo scope della Servlet e le operazioni dei due metodi.

Un **esempio** più pratico del corretto formato lo vediamo nel frammento di codice seguente:

```java
package src;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *  Descrizione di cosa fa la Servlet
 */

@WebServlet("/ExampleServlet")
public class ExampleServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;


    public ExampleServlet() {
    }

    /**
     * Descrizione di cosa fa il doGet
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        /* Content of GET */
    }

    /**
     * Descrizione di cosa fa il doPost
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        /* Content of POST */
    }

}
```

È possibile aggiungere qualsiasi clausola si voglia all'interno del codice Javadoc della classe e dei metodi. Inoltre, è possibile anche implementare metodi o funzioni proprie della classe in modo da rendere le sue operazioni più modulari.

1.3 Definizioni, acronimi e abbreviazioni

Acronimi:

- **SDD:** System Design Document
- **ODD:** Object Design Document
- **RAD:** Requirements Analysis Document

Abbreviazioni:

- **DB:** Database
- **DBMS:** Database Management System

Definizioni:

- **Servlet:** Classi ed oggetti Java per la gestione di operazioni su un Web Server

1.4 Riferimenti

Il contesto è ripreso dal **RAD** e dall' **SDD** del progetto YouLearn.

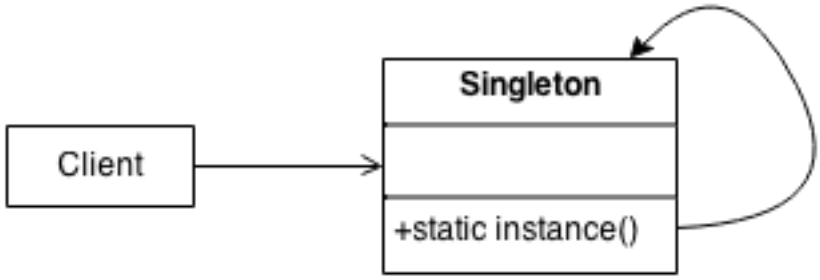
È stato anche usato come riferimento il libro:

Object-Oriented Software Engineering:
Using UML, Patterns, and Java, 3rd Edition
Prentice Hall, Upper Saddle River, NJ, September 25, 2009.

Infine, sono stati usati dei materiali di supporto visionabili al link:
https://wwwbruegge.in.tum.de/lehrstuhl_1/component/content/article/217-OOSE

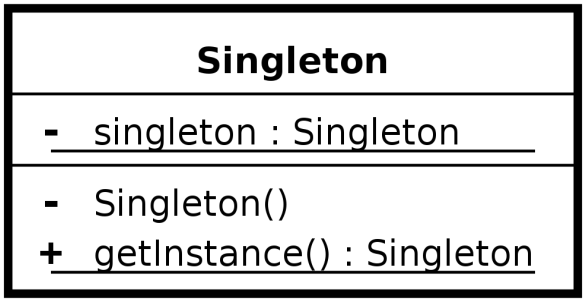
2. Design Pattern

Utilizzeremo il Singleton pattern per le classi di tipo Manager.



Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata *una e una sola* istanza, e di fornire un punto di accesso globale a tale istanza.

L'implementazione più semplice di questo pattern prevede che la classe *singleton* abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce in un metodo getter statico che restituisce l'istanza della classe (sempre la stessa), creandola preventivamente o alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico.



2. Package

La gestione del nostro sistema è suddivisa in tre livelli (three-tier):

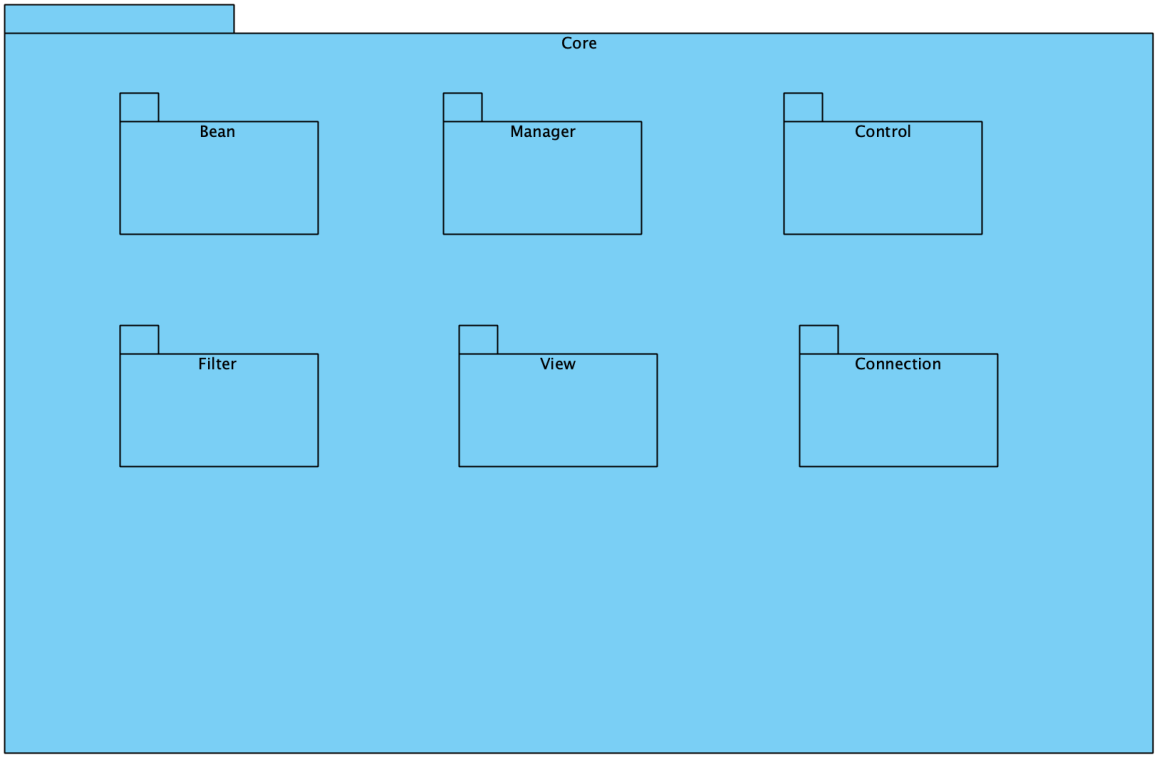
- Interface layer
- Application Logic layer
- Storage layer

Il package "com" contiene i sottopackage che a loro volta inglobano classi atte alla gestione delle richieste utente. Le classi contenute nel package svolgono il ruolo di gestore logico del sistema.

Interface layer	Rappresenta l'interfaccia del sistema, ed offre la possibilità all'utente di interagire con quest'ultimo, offrendo sia la possibilità di inviare, in input, che di visualizzare, in output, dati.
Application Logic layer	Ha il compito di elaborare i dati da inviare al client, e spesso grazie a delle richieste fatte al database, tramite lo Storage Layer, accede ai dati persistenti. Si occupa di varie gestioni quali: 1. Gestione Utente 2. Gestione Corsi 3. Gestione Lezioni 5. Gestione Pagamento 6. Gestione Mail
Storage layer	Ha il compito di memorizzare i dati sensibili del sistema, utilizzando un DBMS, inoltre riceve le varie richieste dall' Application Logic layer inoltrandole al DBMS e restituendo i dati richiesti.

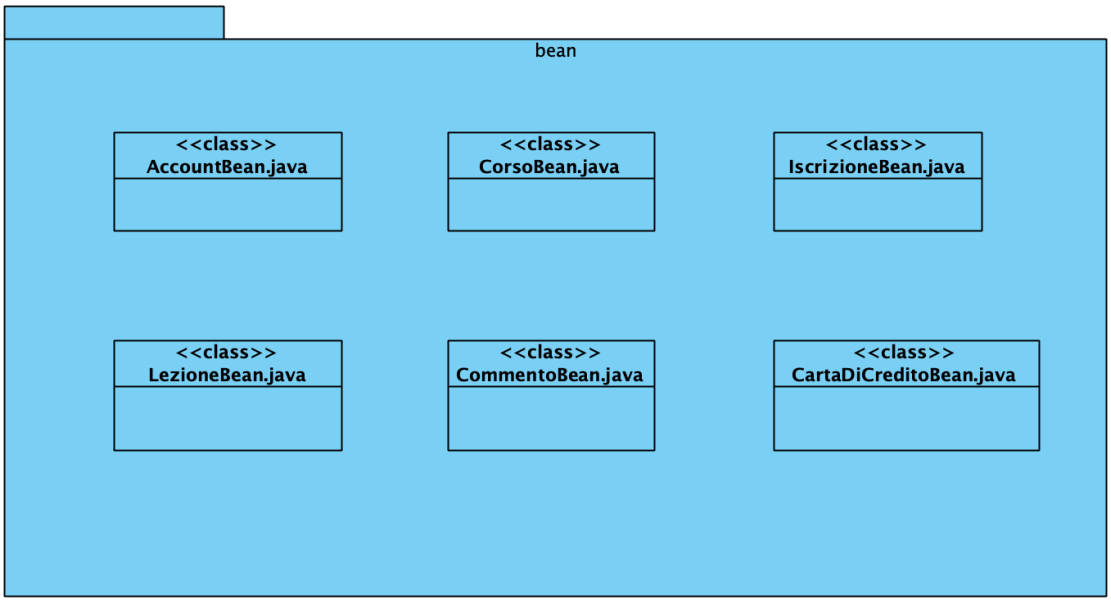
3.1 Package core

In questa sezione verrà illustrata la schematizzazione dei package presenti all'interno del progetto. Tali package verranno divisi a secondo delle operazioni che eseguiranno all'interno del sistema, non saranno presenti i componenti per le funzionalità che non saranno implementate all'interno di questa versione del software. Di seguito ecco riportato il package core che raggruppa tutti i sottopackage funzionali che verranno utilizzati nella fase di implementazione:



3.1.1 Package Bean

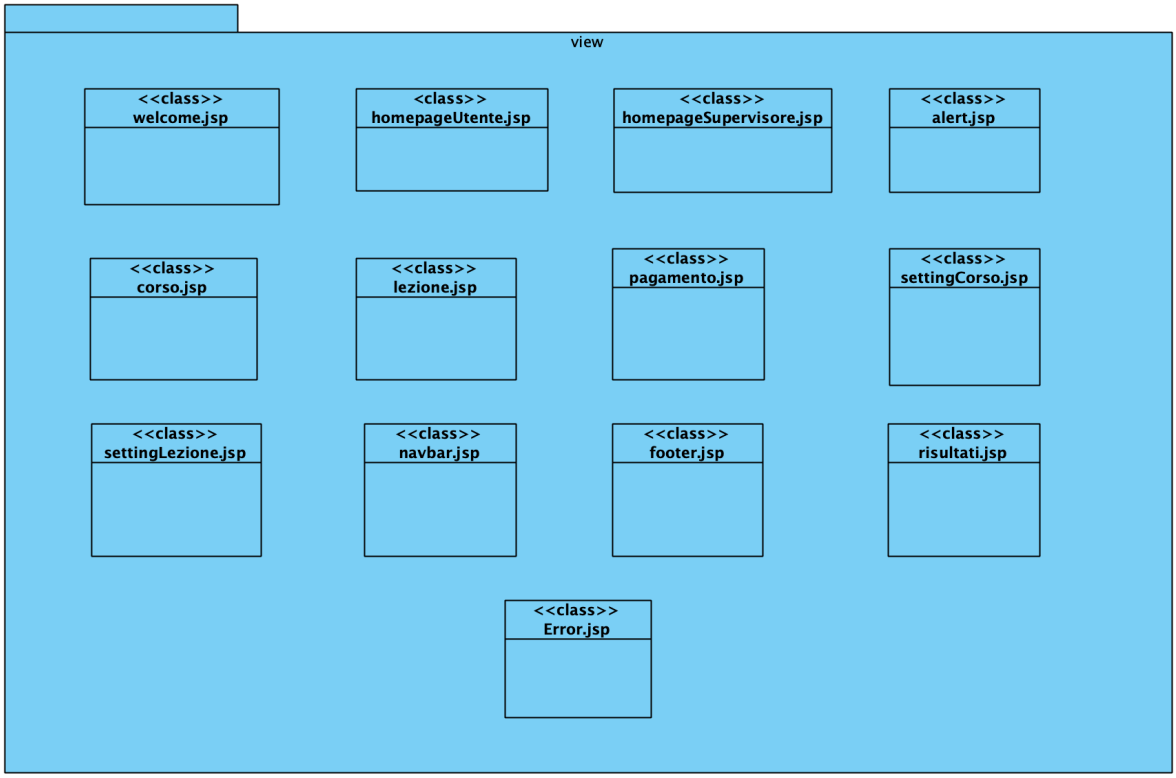
Il package bean include tutte le classi JavaBean. Di seguito, ecco riportato il package che raggruppa tutte le classi del sistema che hanno tale caratteristica:



Nome:	Descrizione:
AccountBean.java	Descrive un account del sistema.
CorsoBean.java	Descrive un corso del sistema.
IscrizioneBean.java	Descrive un'iscrizione di un utente verso un corso del sistema.
LezioneBean.java	Descrive una lezione propria di un corso del sistema.
CommentoBean.java	Descrive un commento di un utente del sistema associato ad una lezione di un corso.
CartaDiCreditoBean.java	Descrive una carta di credito affiliata ad un utente del sistema

3.1.2 Package View

Il package view include tutte le pagine JSP, ossia quelle pagine che gestiscono la visualizzazione dei contenuti da parte di un utente del sistema. Di seguito, ecco riportato il package che raggruppa tutte le pagine del sistema che hanno tale caratteristica:

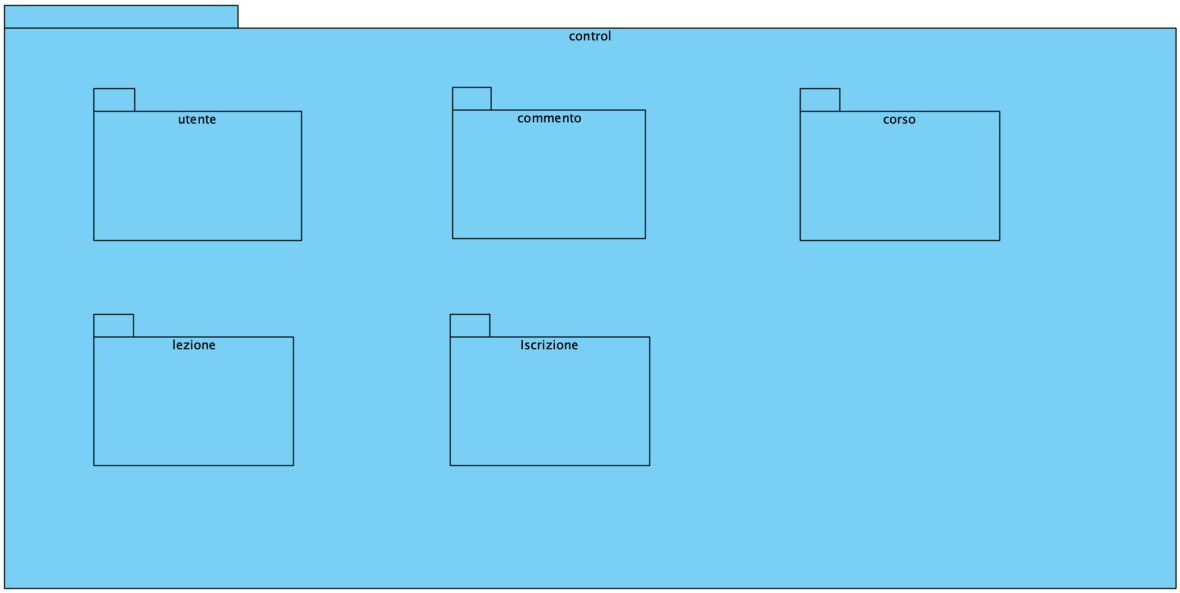


Nome:	Descrizione:
welcome.jsp	Pagina di presentazione della piattaforma, presenta il form di login e di registrazione.
homepageUtente.jsp	Pagina principale della piattaforma per l'utente, presenta le informazioni personali dell'account con i relativi tool di modifica e informazioni inerenti ai corsi relazionati con l'utente.
homepageSupervisore.jsp	Pagina principale della piattaforma per il supervisore, presenta le informazioni personali dell'account con i relativi tool di modifica e informazioni inerenti ai corsi relazionati con il supervisore.

risultati.jsp	Pagina che presenta i risultati di una ricerca secondo una forma tabellare.
alert.jsp	Pagina che presenta un avviso all'interno di una finestra in seguito ad un'azione con conferma. Comprende anche le pagine di decisione del supervisore.
corso.jsp	Pagina principale dei corsi, presenta tutte le informazioni legate ad un corso compreso la lista delle lezioni.
lezione.jsp	Pagina principale della lezione, presenta un video e la lista dei commenti affiliata.
pagamento.jsp	Pagina che presenta un form di pagamento da compilare per l'acquisto di un corso.
settingCorso.jsp	Pagina di modifica o di creazione di un corso.
settingLezione.jsp	Pagina di modifica, cancella o aggiunge una lezione.
navbar.jsp	Pagina che include la barra di navigazione principale del sistema.
footer.jsp	Pagina che include il footer delle pagine del sistema.
Error.jsp	Pagina di errore in seguito ad un'operazione che non va a buon fine o ad una pagina non reperibile.

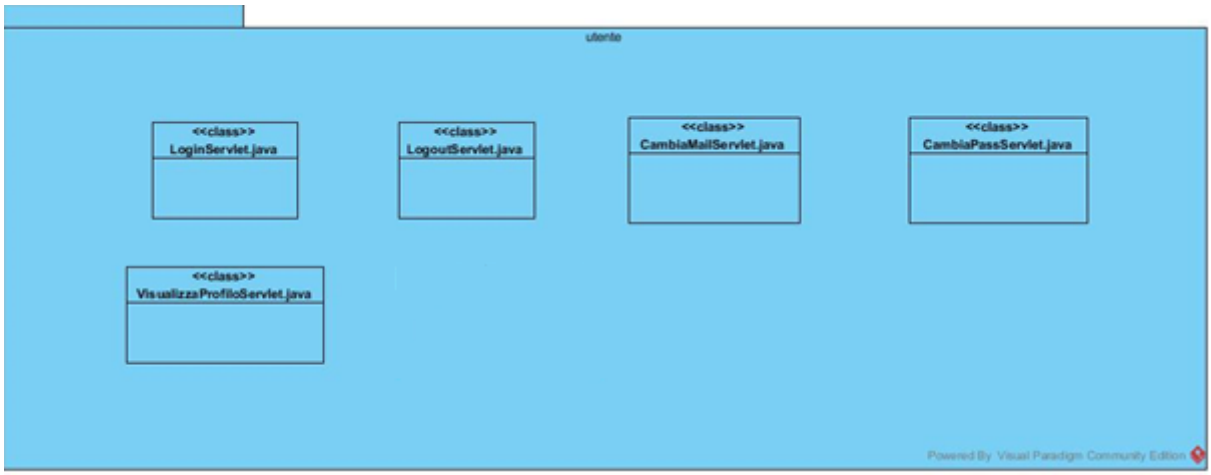
3.1.3 Package Control

Il package control include tutte le classi Servlet che rappresentano la logica applicativa della web platform. Il sistema vede la separazione di tali classi in sottopackage differenziati dai vari sottosistemi della web platform proposta. Di seguito, illustriamo la suddivisione dei sottopackage con il seguente diagramma:



3.1.3.1 Package Utente

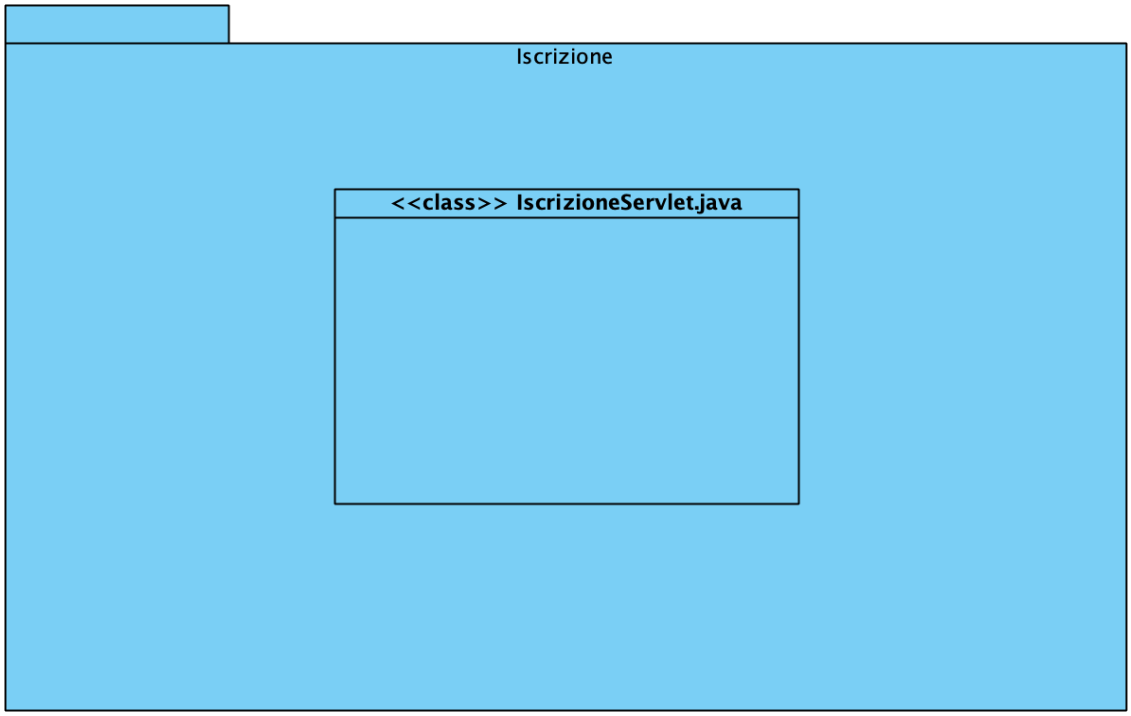
Il package utente include tutte quelle classi Servlet adibite a svolgere una funzionalità del sottosistema di gestione dell'utente. Di seguito presentiamo il contenuto di tale package con il seguente diagramma:



Nome:	Descrizione:
LoginServlet.java	Controller che permette di completare un'operazione di login.
LogoutServlet.java	Controller che permette di effettuare il log out.
CambiaMailServlet.java	Controller che permette di cambiare l'e-mail di registrazione.
CambiaPassServlet.java	Controller che permette di cambiare la password dell'account.
VisualizzaProfiloServlet.java	Controller che permette di recuperare informazioni sul profilo dell'Account in sessione

3.1.3.2 Package Iscrizione

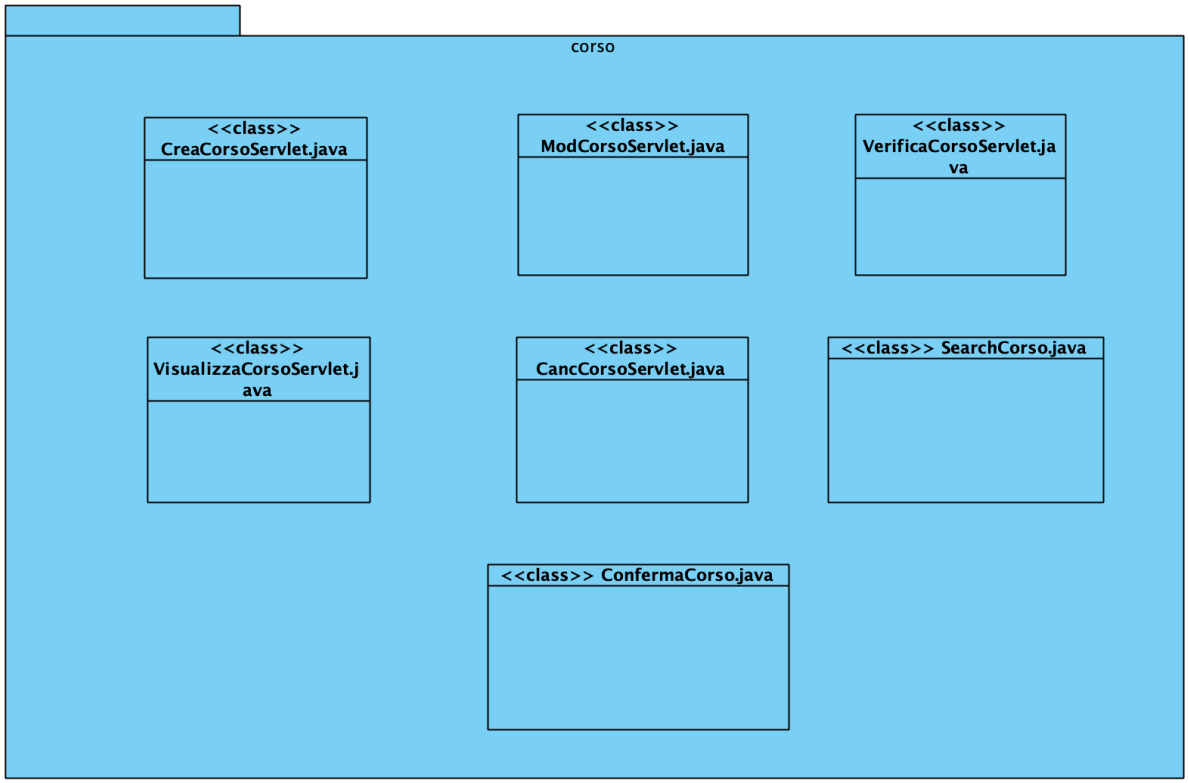
Il package utente include tutte quelle classi Servlet adibite a svolgere una funzionalità del sottosistema di gestione delle iscrizioni.



Nome:	Descrizione:
IscrizioneServlet.java	Controller che permette di gestire le iscrizioni ad un corso di uno studente.

3.1.3.3Package Corso

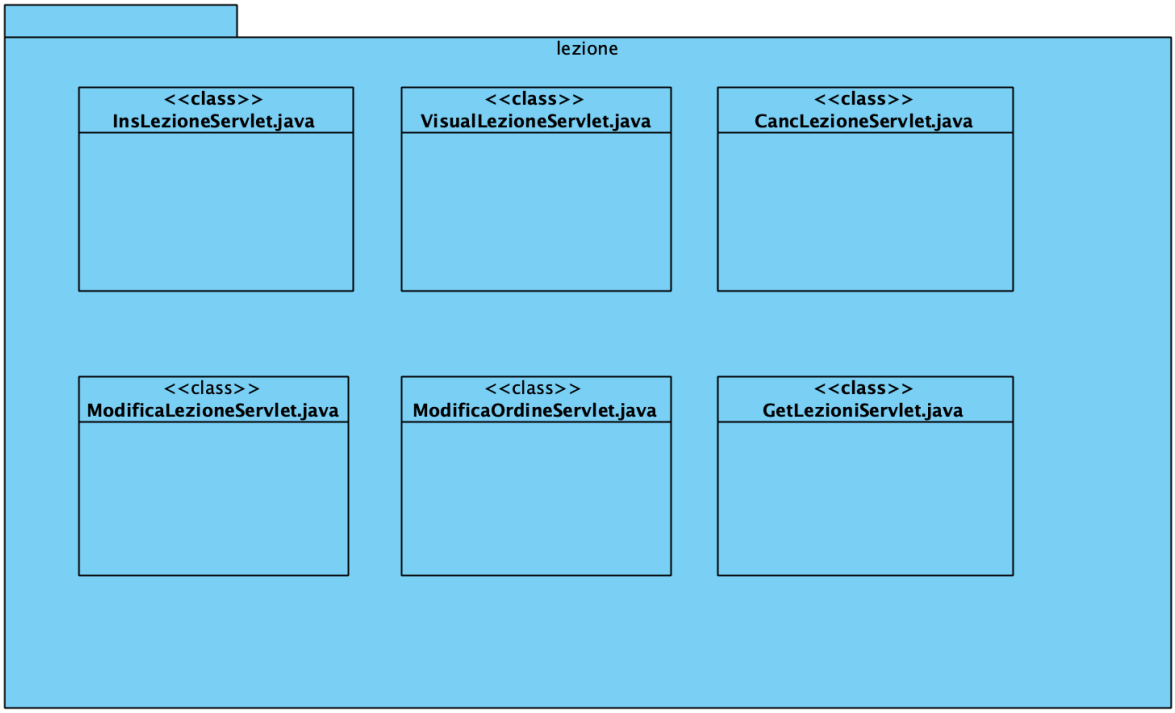
Il package utente include tutte quelle classi Servlet adibite a svolgere una funzionalità del sottosistema di gestione dei corsi. Di seguito presentiamo il contenuto di tale package con il seguente diagramma:



Nome:	Descrizione:
CreaCorsoServlet.java	Controller che gestisce la creazione di un corso.
ModCorsoServlet.java	Controller che gestisce le modifiche di un corso.
VerificaCorsoServlet.java	Controller che accetta o rifiuta un corso visto da un supervisore.
VisualizzaCorsoServlet.java	Controller che gestisce la visualizzazione di un corso.
CancCorsoServlet.java	Controller che gestisce la cancellazione di un corso.
SearchCorso.java	Controller che gestisce la ricerca di un corso.
ConfermaCorso.java	Controller che gestisce la conferma di un corso in fase di completamento.

3.1.3.4 Package Lezione

Il package utente include tutte quelle classi Servlet adibite a svolgere una funzionalità del sottosistema di gestione delle lezioni. Di seguito presentiamo il contenuto di tale package con il seguente diagramma:

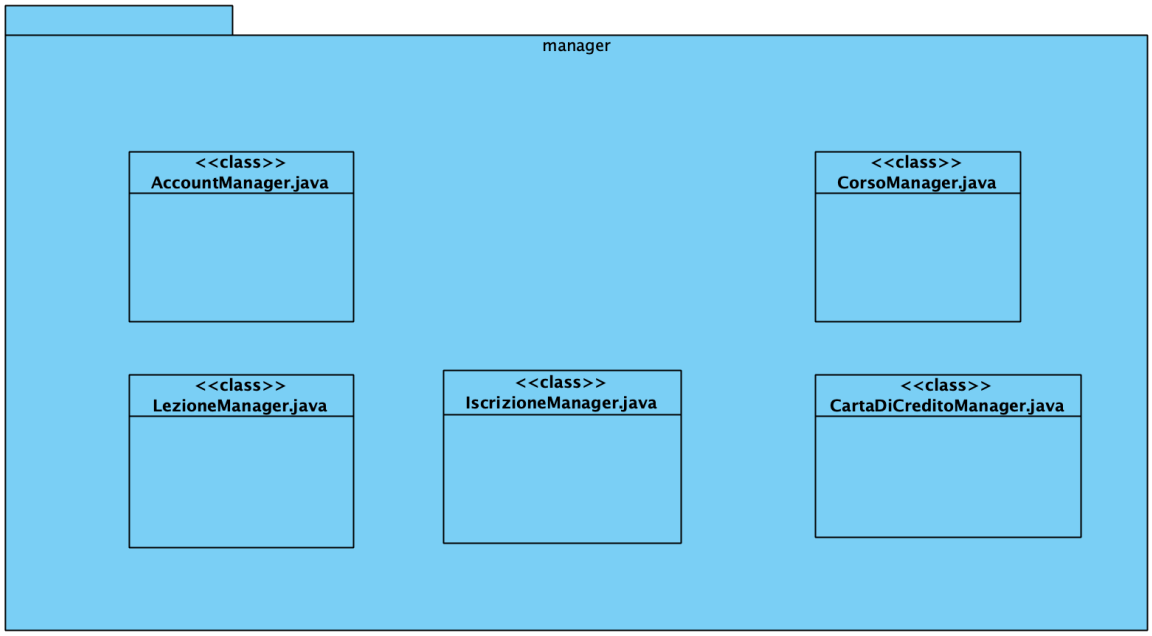


Nome:	Descrizione:
InsLezioneServlet.java	Controller che gestisce l'inserimento di una lezione.
CancLezioneServlet.java	Controller che gestiscono la cancellazione di una lezione.
VisualLezioneServlet.java	Controller che gestisce la visualizzazione di una lezione
ModificaLezioneServlet.java	Controller che modifica i dati di una lezione di un corso in fase di completamento.
ModificaOrdineServlet.java	Controller che gestisce l'ordine delle lezioni di un corso in fase di completamento.
GetLezioneServlet.java	Controller che preleva e visualizza le lezioni di un corso in fase di completamento, solamente il docente di un corso riesce ad

	utilizzare in maniera legale tale controller.

3.1.4 Package Manager

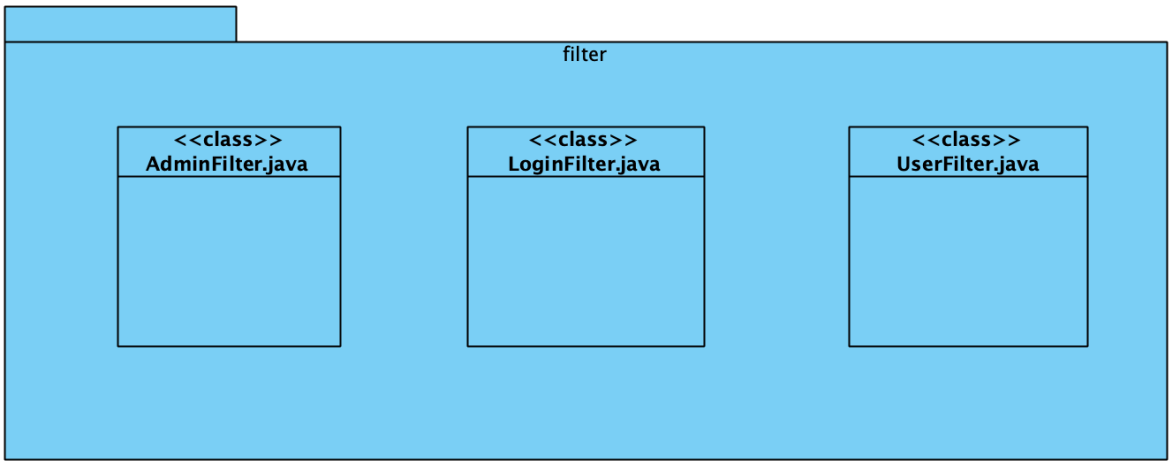
Il package Manager include tutte le classi Java adibite all'accesso dei dati presenti nel sistema. Di seguito, ecco riportato il package che raggruppa tutte le classi del sistema che hanno tale caratteristica:



Nome:	Descrizione:
AccountManager.java	Classe di gestione dei dati di un account.
CorsoManager.java	Classe di gestione dei dati di un corso.
LezioneManager.java	Classe di gestione dei dati di una lezione e dei commenti affiliati.
IscrizioneManager.java	Classe di gestione dati dell'iscrizione
CartaDiCreditoManager.java	Classe di gestione dei dati di una carta di credito.

3.1.5 Package Filter

Il package filter include tutte le classi adibite al filtraggio dei dati. Di seguito, ecco riportato il package che raggruppa tutte le classi del sistema che hanno tale caratteristica:



Nome:	Descrizione:
AdminFilter.java	Filtro Java che preclude l'accesso tranne ai supervisori.
LoginFilter.java	Filtro Java che preclude l'accesso tranne agli utenti loggati.
UserFilter.java	Filtro Java che divide le viste tra utente e supervisore.

4.Class interfaces

I contratti rappresentano le pre-condizioni, post-condizioni e le invarianti delle classi Manager progettate per compiere funzionalità atomiche legate ai servizi dei sottosistemi specificati nel documento di System Design.

4.1 Manager di account

Nome Classe	AccountManager
Descrizione	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione Utente in maniera persistente.
Pre-condizioni	**context** AccountManager::doRetrieveByKey(key): **pre: key! =null** **context** AccountManager::modificaPassword(mail,password): **pre: mail!=null && password!=null** && checkMail(mail) **context** AccountManager::modificaMail(mail, newMail): **pre: mail!=null && newMail != null** && checkMail(mail) && !checkMail(newMail) **context** AccountManager::login(mail,password): **pre: mail!=null && password!=null** **context** AccountManager::checkMail(mail): **pre: mail!=null** **context** AccountManager::checkAccount(account): **pre: account! =null** **context** AccountManager::isWellFormatted(account): **pre: account! =null**
Post-condizioni	**context** AccountManager::modificaPassword(mail,password): **post: doRetrieveByKey(mail).getPassword() == password** **context** AccountManager::modificaMail(mail,newMail) **post: !checkMail(mail) && checkMail(newMail)**
Invarianti:	

4.2 Manager di Carta di credito

Nome Classe:	CartaDiCreditoManager
Descrizione:	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema di Gestione Utente e Gestione corsi in maniera persistente.
Pre-condizione:	**context** CartaDiCreditoManager::doRetrieveByKey(key): **pre: key! =null** **context** CartaDiCreditoManager::registerCard(carta): **pre: carta!= null &&** isWellFormatted(carta) && !checkCarta(carta.getNumeroCarta()) **context** CartaDiCreditoManager::modifyCard(newCarta, numeroCarta) **pre: newCarta==null** && isWellFormatted(newCarta) && checkCarta(numeroCarta)&& !checkCarta(newCarta.getNumeroCarta()) **context** CartaDiCreditoManager::checkCarta(numeroCarta) **pre: numeroCarta != null** **context** CartaDiCreditoManager::retrieveByAccount(account) **pre:** AccountManager.checkAccount(account) && account.getTipo().equals(Ruolo.Utente) **context** CartaDiCreditoManager::isWellFormatted(carta) **pre: carta!=null**
Post-condizione:	**context** CartaDiCreditoManager::registerCard(carta): **post: checkCarta(carta.getNumeroCarta())** **context** CartaDiCreditoManager::modifyCard(newCarta, numeroCarta) **post: !checkCarta(numeroCarta)&&** **checkCarta(newCarta.getNumeroCarta())**
Invarianti:	

4.3 Manager di Corso

Nome Classe:	CorsoManager
Descrizione:	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema per la gestione dei corsi in maniera persistente.
Pre-condizione:	**context** CorsoManager::doRetrieveByKey(key): **pre: key! =null** **context** CorsoManager::searchCorso(key): **pre:** key!=null&& !key.equals(“”) **context** CorsoManager::creaCorso(corso, copertina) **pre:** corso != null && corso.getIdCorso()!=null && copertina != null && isWellFormatted(corso) && accountManager.checkMail(corso.getDocente().getMail()) && !checkCorso(corso) **context** CorsoManager::doUpdate(corso)

	pre: corso != null **context** CorsoManager::modificaCorso(corso, file) **pre: corso != null && file != null** && checkCorso(corso.getIdCorso())&& isWellFormatted(corso) && corso.getStato.equals(Stato.Completamento) **context** CorsoManager::removeCorso(id) **pre: checkCorso(id) && id >= 0** **context** CorsoManager::convalidaCorso(value, corso) **pre: value!=null** && isWellFormatted(corso) && corso.getIdCorso()) != null && checkCorso(corso.getIdCorso()) && corso.getStato().equals(Stato.Attesa) **context** CorsoManager::confermaCorso(corso) **pre:** corso != null && isWellFormatted(corso) && corso.getStato().equals(Stato.Completamento) && corso.getIdCorso() != null && checkCorso(corso.getIdCorso()) **context** CorsoManager::retrieveByCreatore (account) **pre: account != null** && AccountManager.checkAccount(account) && account.getTipo().equals(Ruolo.Utente) **context** CorsoManager::doRetrieveBySupervisore (account) **pre: account != null** && AccountManager.checkAccount(account) && account.getTipo().equals(Ruolo.Supervisore) **context** CorsoManager::checkCorso (corso) **pre: corso != null** **context** CorsoManager::checkCorso (id) **pre: id >= 0** **context** CorsoManager::isWellFormatted(corso) **pre: corso != null**
Post-condizione:	**context** CorsoManager::creaCorso(corso, copertina) **post: checkCorso(corso)** **context** CorsoManager::convalidaCorso(value, corso) **post: corso.getStato.equals(Stato.Attivo)** || **corso.getStato.equals(Stato.Completamento)** **context** CorsoManager::confermaCorso(corso) **post: corso.getStato.equals(Stato.Attesa)**
Invarianti:	

4.4 Manager di Lezione

Nome Classe:	LezioneManager
Descrizione:	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema per la gestione lezioni in maniera persistente.
Pre-condizione:	**context** LezioneManager::ModificaOrdine(id, coppie) **pre: id>=0 && CorsoManager.checkCorso(id)** && coppie != null && coppie.matches("^([0-9]+[-][0-9]+[,])*([0-9]+[-][0-9]+)$") **context** LezioneManager::changeNumeroLezione(lezione, connection): **pre: connection!= null && lezione != null && lezione.getNumeroLezione() >= 0 && lezione.getIdLezione()!=null** **context** LezioneManager::insLezione(lezione, file): **pre: lezione != null && !checkLezione(lezione.getIdLezione()) && file!=null** && lezione.getCorso().getIdCorso()!=null && CorsoManager.checkCorso(lezione.getCorso().getIdCorso()) **context** LezioneManager::delLezione(lezione): **pre:** checkLezione(lezione.getIdLezione()) **context** LezioneManager::checkLezione(id) **pre: id >= 0** **context** LezioneManager::checkLezione(lezione) **pre:** lezione != null && lezione.getIdLezione()!= null && isWellFormatted(lezione) **context** LezioneManager::retrieveCommentoById(id): **pre: id>=0** **context** LezioneManager::retrieveCommentiByLezione(lezione): **pre: lezione !=null &&** lezioneIsWellFormatted(lezione) **context** LezioneManager::delCommento(id): **pre: id>=0** && checkCommento(id) **context** LezioneManager::insCommento(commento): **pre:** commento!=null && **!checkCommento(commento.getIdCommento())** && checkLezione(commento.getLezione().getIdLezione()) **context** retrieveLezioniByCorso(corso): **pre: corso!=null** && corso.getIdCorso()!=null && CorsoManager.isWellFormatted(corso) **context** LezioneManager.checkCommento(key) **pre: key != null**
Post-condizione:	**context** LezioneManager::insLezione(lezione, file): **post: checkLezione(lezione.getIdLezione())** **context** LezioneManager::delLezione(lezione): **post: !checkLezione(lezione.getIdLezione())** **context** LezioneManager::delCommento(id): **post: !checkCommento(id)**

	context LezioneManager::insCommento(commento): **post: checkCommento(commento.getIdCommento())** **context** LezioneManager::insLezione(lezione, file): **post: checkLezione(lezione.getIdLezione())**
Invarianti:	

4.5 Manager di Iscrizione

Nome Classe:	IscrizioneManager
Descrizione:	Classe che gestisce alcune funzionalità legate ai servizi del sottosistema per la gestione dei corsi in maniera persistente
Pre-condizione:	**context** IscrizioneManager::doRetrieveByKey(idCorso, mail): **pre: idCorso>=0 && mail!=null** **context** IscrizioneManager::getIscrizioniUtente(account): **pre: account!=null && AccountManager.checkMail (account.getMail()) && account.getTipo().equals(Ruolo.Utente)** **context** IscrizioneManager::getIscrittiCorso(corso): **pre: corso!=null** && CorsoManager.checkCorso(corso) **context** IscrizioneManager::iscriviStudente(iscrizione): **pre:** isWellFormatted(iscrizione) && !checkIscrizione(iscrizione.getCorso().getIdCorso(), iscrizione.getAccount.getMail()) && CorsoManager.checkCorso(iscrizione.getCorso()) && AccountManager.checkAccount(iscrizione.getAccount()) **context** IscrizioneManager::isWellFormatted(iscrizione) **pre: iscrizione != null** **context** IscrizioneManager::checkIscrizione(id,mail): **pre: id>=0 && mail!=null**
Post-condizione:	**context** IscrizioneCorso::iscriviStudente(iscrizione): **post: checkIscrizione(iscrizione.getCorso().getIdCorso(), iscrizione.getAccount.getMail())**
Invarianti:	

5.Glossario

Astrazioni: Concetto che, in informatica, astrae l'implementazione per concentrarsi sulla progettazione strutturale di un oggetto

Modularità: Progettazione di un sistema basato sullo sviluppo delle sue attività in componenti indipendenti collegate tra di loro.

Java Bean: Classi Java che hanno il compito di incapsulare altri oggetti in modo da facilitare la gestione del trasferimento dei dati del sistema.

Java Control: Classi Java che hanno il compito di gestire la logica applicativa del sistema

Java Manager: Classi Java che hanno il compito di gestire l'accesso e le modifiche ai dati persistenti del sistema.

JSP: Java Servlet Page, hanno il compito di gestire l'interfacciamento dell'utente con il sistema

Package: Insieme di classi, pagine o altri tipi di elementi che vengono raggruppati secondo una particolare logica.

Filter: Componenti in grado di filtrare i dati e gestirli secondo alcune condizioni specifiche implementate.