Laurea Triennale in Informatica - Università di Salerno
Corso di *Ingegneria del Software* - Prof.ssa F. Ferrucci

Object Design Document

EasyPlan

Riferimento	
Versione	1.0
Data	16/12/2018
Destinatario	Prof.ssa F. Ferrucci
Presentato da	Giuseppe Abagnale Roberto Gagliardi
Approvato da	

Sommario

Revision History

Data	Versione	Descrizione	Autori
16/12/2018	0.1	Prima stesura	Francesco Aurilio Francesca Cerruto Carmine D'Angelo Andrea Junquera Pier Giuseppe Rotondo Francesco Tranzillo Emanuele Vitale
16/12/2018	1.0	Revisione prima stesura	Giuseppe Abagnale Roberto Gagliardi

Object Design Document del Progetto EasyPlan

1. Introduzione

Il documento che segue illustra diversi dettagli legati alla fase implementativa del sistema *EasyPlan*; in particolare, esso descrive i trade-off di progettazione definiti dagli sviluppatori, le linee guida da seguire per le interfacce dei sottosistemi, la decomposizione dei sottosistemi in pacchetti e classi e, infine, la specifica delle interfacce delle classi.

1.1. Trade-off

Efficienza vs Portabilità

Il sistema è progettato per far in modo che broswer diversi e dispositivi aventi risoluzioni diverse (come smartphone, laptop o computer desktop) possano visualizzare correttamente le pagine web del sito *EasyPlan* sfruttando al meglio lo spazio del display. Mettendo al primo posto la portabilità, il sistema non garantisce lo stesso livello di efficienza di un sistema progettato per un solo tipo di dispositivo target, poiché una tale adattabilità richiederebbe un carico maggiore da gestire, specialmente a causa dell'elevata dimensione dei fogli di stile (file css).

Comprare vs Costruire

Utilizzare software già realizzato da qualcun altro permette di avvalersi di diversi vantaggi, tra cui l'utilizzo di funzionalità già complete e una minore quantità di lavoro per gli sviluppatori; tuttavia, questo può portare anche un notevole aumento dei costi e un impegno non indifferente nell'integrare le componenti già realizzate con quelle costruite dagli sviluppatori. Per questi motivi, è stato deciso di realizzare la maggior parte del sistema partendo da zero, utilizzando delle componenti esterne soltanto in alcuni casi. Questi includono delle librerie da integrare con i linguaggi utilizzati per la generazione dei file pdf e altri strumenti esterni per semplificare la popolazione iniziale del database di sistema.

Memoria vs Efficienza

Nella fase di system design si è preferito dare maggiore priorità ad aspetti come l'usabilità e l'affidabilità, le quali prevedono, in particolare, la possibilità di inserire una significativa quantità di dati ridondanti all'interno del database. In questo caso, per semplificare il processo di interrogazione del database, sarebbe necessario introdurre ulteriore ridondanza di dati. Per questo motivo, si è deciso di evitare un ulteriore aumento di carico della memoria aumentando però la complessità di composizione delle query.

1.2. Componenti off-the-shelf

Per lo sviluppo del sistema è previsto l'uso di diversi componenti off-the-shelf, ovvero componenti software messi a disposizione dal mercato che offrono pacchetti di soluzioni che possono essere utili a risolvere degli specifici problemi.

Le componenti previste per la realizzazione del sistema sono le seguenti:

- **Bootstrap**, un toolkit open source utilizzato per lo sviluppo di progetti responsive sul web. Il suo utilizzo è previsto insieme a quello di HTML, CSS e JavaScript per realizzare la struttura base della grafica.
- **jQuery.js**, una libreria JavaScript per applicazioni web il cui obiettivo è quello di semplificare la selezione, la manipolazione, la gestione degli eventi e l'animazione di elementi DOM in pagine HTML. Il suo utilizzo ha lo scopo di ridurre la complessità del codice JavaScript durante l'implementazione.
- **jspdf.js** e **pdfFromHTML.js**, due librerie javascript che consentono di trasformare codice HTML in un documento in formato PDF. Il loro utilizzo serve a convertire il piano di studi, compilato sullo studente su EasyPlan, in un file PDF scaricabile.
- **Javadoc**, un tool che permette di generare la documentazione di un programma attraverso l'inserimento di tag specifici nel codice stesso; inoltre, prevede che la documentazione in javadoc produca un insieme di pagine HTML consultabili sul web. Il tool viene utilizzato per documentare il codice Java scritto dagli sviluppatori così da garantire maggiore comprensibilità del codice e rendere più facile la sua manutenzione a sviluppatori futuri.
- **AJAX**, una tecnica di sviluppo software per la realizzazione di applicazioni web interattive che si basa su uno scambio di dati in background fra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente. Il suo utilizzo consente di effettuare richieste asincrone al database.
- **Tabula**, un tool utilizzato per prelevare tabelle di dati da file pdf ed estrapolarli in file in formato CVS o fogli elettronici Microsoft Excel. Il suo utilizzo permette di prelevare i dati interessati dalla Guida dello Studente e di velocizzare quindi la popolazione del database.
- **SQLizer**, un tool disponibile online che permette di convertire i nostri documenti in script MySQL. Il suo utilizzo consente di semplificare l'inserimento dei dati nel database, grazie alla generazione automatica di query (INSERT) estrapolate dal documento in formato CVS o Microsoft Excel generato da Tabula.

1.3. Linee guida per la documentazione dell'interfacce

Il progetto verrà realizzato con l'IDE di sviluppo Eclipse e sarà strutturato nel seguente modo:

- Il progetto viene suddiviso in tre package (Model, Control, View), i quali contengono i rispettivi sub-package con le rispettive classi.
- Le classi Java, oltre a prevedere le convenzioni più comuni del linguaggio, presentano il seguente schema:
    1. Clausole import.
    2. Dichiarazione della classe.
    3. Variabili di istanza.
    4. Costruttore.
    5. Metodi.
- Il nome di una classe deve rispettare il seguente formato: *NomeClasse*.
- Il nome di un metodo o di una variabile di istanza deve rispettare il seguente formato: *nomeDellaVaribile, nomeDelMetodo.*
- L'insieme delle variabili d'istanza è preceduto e seguito da una riga vuota.
- Un intero metodo, compreso di intestazione ed istruzioni, è preceduto e seguito da una riga vuota.
- I nomi utilizzati per identificare classi, attributi e metodi devono risultare quanto più possibile significativi del loro scopo.
- Una parentesi graffa aperta è preceduta da uno spazio e seguita da un carattere di new line che comincia con un carattere di indentazione.
- Una parentesi graffa chiusa è preceduta e seguita da un carattere di new line.
- Un costrutto di tipo *if* che prevede una sola istruzione può evitare l'uso delle parentesi graffe.
- I commenti possono essere utilizzati soltanto nei casi ritenuti opportuni. Nel caso in cui un commento si estenda per una sola riga, esso presenta il formato: // commento; se, invece, un commento si estende su più righe diverse, esso presenta il formato /* commento */.
- Ogni classe e ogni metodo devono essere corredate da commenti che rispettano lo standard utilizzato da Javadoc per la produzione di documentazione in formato HTML.
- Il package Model contiene tutte le classi che fanno riferimento ad entità persistenti (Bean e DAO).
- Il package Control contiene tutte le classi Servlet (@WebService) che si occupano della logica di business del sistema e agiscono da interlocutore tra le classi contenute nel package Model e le classi contenute nel package View.

- Il package View contiene tutti i file che si occupano dell'interfaccia utente (JSP, pagine HTML).
- Dov'è possibile, in modo particolare per i Bean contenuti nel package Model, le classi sono corredate da opportuni metodi GETTER, SETTER e, eventualmente, sovrascrivono il metodo *toString()*.
- Una singola indentazione consiste di un solo carattere di tabulazione e, qualunque sia il linguaggio usato per la produzione di codice, ogni istruzione risulta opportunamente indentata.
- Le dichiarazioni si trovano sempre all'inizio dei corrispondenti blocchi di codice, poiché dichiarare le variabili soltanto in corrispondenza del loro primo utilizzo può provocare confusione.
- Qualsiasi variabile presente in un blocco più interno non è mai dichiarata con lo stesso nome di una variabile presente in un blocco più esterno.

1.4. Design Pattern

Di seguito, vengono descritti i design pattern ritenuti idonei all'implementazione del sistema proposto.

1.4.1. Model - View - Controller

Siccome in fase di system design si è stabilito che il sistema proposto presenta l'architettura Model - View - Controller (o MVC), in fase di implementazione è previsto l'utilizzo dell'omonimo pattern.
Il pattern MVC viene utilizzato in un contesto dove l'applicazione deve fornire un'interfaccia grafica costituita da più schermate che mostrano vari dati all'utente, i quali devono risultare aggiornati in qualunque momento. Tale pattern viene spesso utilizzato nei casi in cui l'applicazione presenti una natura modulare basata sulle responsabilità, al fine di ottenere un sistema basato sulle componenti.
La soluzione fornita da questo pattern risulta essere particolarmente adatta al sistema proposto, poiché prevede che l'applicazione debba separare i componenti software che implementano le funzionalità di business dai componenti che implementano la logica di presentazione e di controllo, i quali utilizzano tali funzionalità.
I componenti previsti dal pattern MVC sono descritti di seguito nel dettaglio:

- Il Model definisce le regole di business per l'interazione con i dati, esponendo alla View ed al Control rispettivamente le funzionalità per l'accesso e l'aggiornamento dei dati.
- Il Control realizza la corrispondenza tra l'input dell'utente e i processi eseguiti dal Model, oltra a selezionare le schermate della View richieste ed implementare la logica di controllo dell'applicazione.

- La View si occupa della logica di presentazione dei dati.

In genere, le View adottano una strategia "push Model", così come prevista dal design pattern dell'Observer. Tuttavia, siccome il presente modello MVC fa riferimento ad un'architettura J2EE, le view che vengono implementate con delle JSP restituiscono GUI costituite da contenuti statici in HTML, non in grado di eseguire delle operazioni sul Model. Per questo motivo, si è deciso di utilizzare la strategia "pull Model", che differisce in alcuni aspetti all'Observer. Questa strategia, infatti, prevede che la View richieda aggiornamenti quando lo ritiene opportuno e deleghi al Control sia l'esecuzione dei processi richiesti dall'utente (dopo averne catturato gli input), sia la scelta delle eventuali schermate da presentare.

1.4.2. Data Access Object

Il pattern Data Access Object (o DAO) è utilizzato per separare l'API di accesso ai dati a basso livello dai servizi di business ad alto livello. In generale, un Pattern DAO segue lo schema seguente.

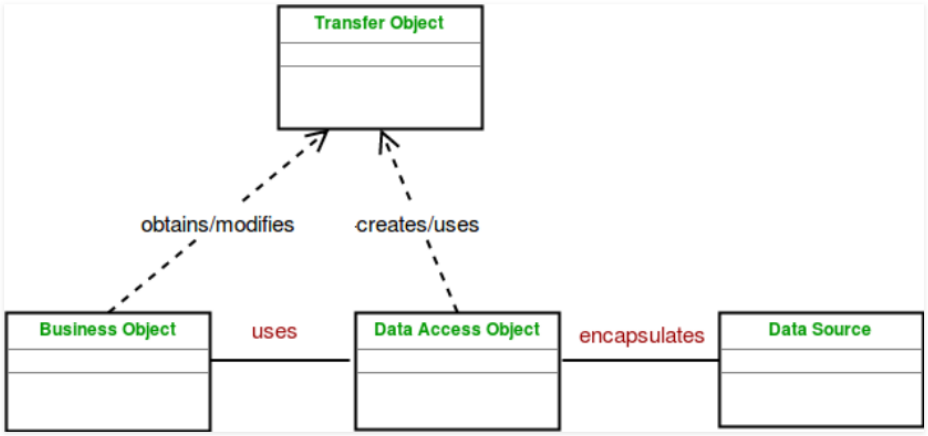

Figura 1 Pattern Data Access Object

Le componenti previste dal pattern DAO sono descritte di seguito:

- Il Business Object rappresenta il client che richiede l'accesso ai dati sorgente così da poterli archiviare. Un Business Object può essere implementato come un bean di sessione, un bean di entità o un altro oggetto Java, oltre a prevedere un servlet o un bean ausiliario che acceda all'origine dei dati.
- Il Data Access Object (DAO) astrae la funzionalità di accesso ai dati sottostanti al Business Object per consentire l'accesso trasparente alla sorgente dei dati. In questo caso, il Business Object delega il caricamento dei dati e memorizza le operazioni sui Data Access Object.
- Il Data Source rappresenta la sorgente dei dati, in questo caso un database relazionale.

- Il Transfer Object rappresenta un oggetto di trasferimento utilizzato come un vettore di dati. Il DAO potrebbe utilizzare un Trasfer Object per restituire i dati al client oppure riceve i dati dal client in un Trasfer Object per aggiornare i dati presenti nel database.

Nel sistema proposto, i Business Object sono rappresentati da Servlet che si servono di Bean e DAO per ricevere e inviare dati da e verso il client. In particolare, i DAO sono impegnati nell'invocazione dei metodi CRUD, mentre il DataSource, rappresentato dalla classe singleton *DriverManagerConnectionPool* (il cui pattern viene illustrato in seguito) ha il compito di gestire la connessione al database che viene poi utilizzata da tutti i DAO per la lettura, la scrittura, l'aggiornamento e l'eliminazione dei dati persistenti nel DBMS MySQL. Per finire, i Transfer Object consistono di tutti gli oggetti Bean che rappresentano le informazioni persistenti che vengono scambiate tra database e client.

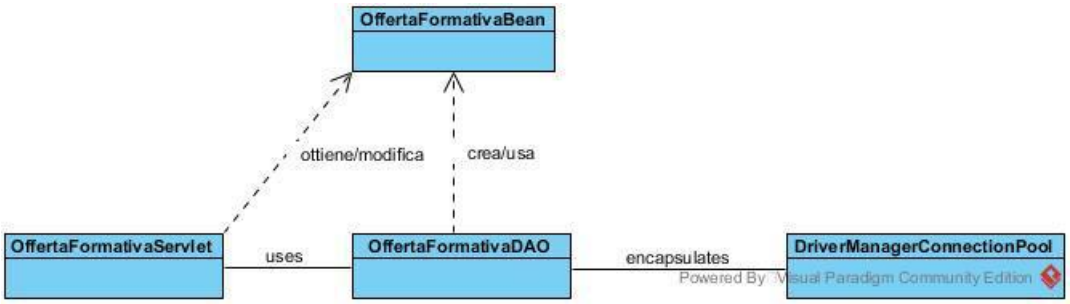

Figura 2 Pattern DAO per offerte formative

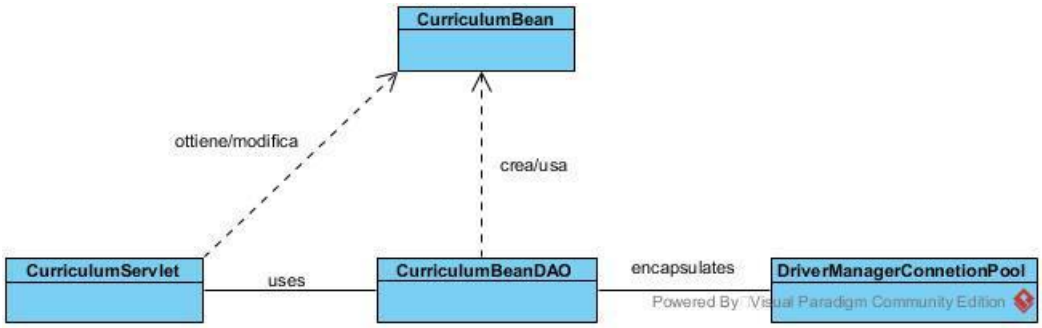

Figura 3 Pattern DAO per curricula

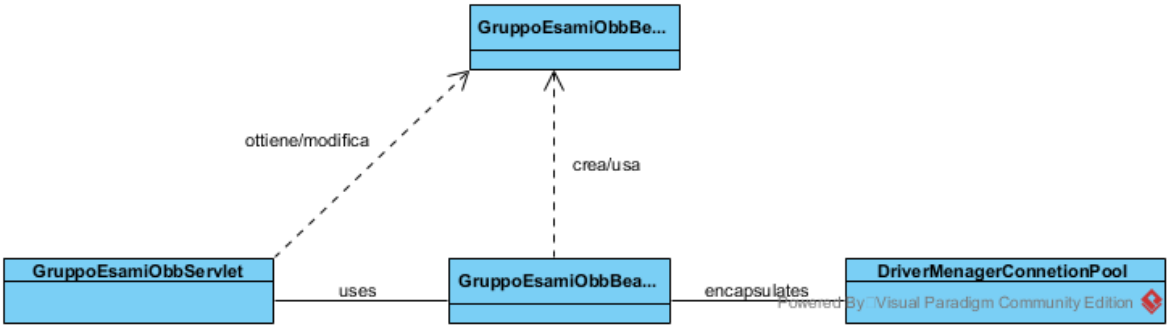

Figura 4 Pattern DAO per gruppi di esami obbligatori

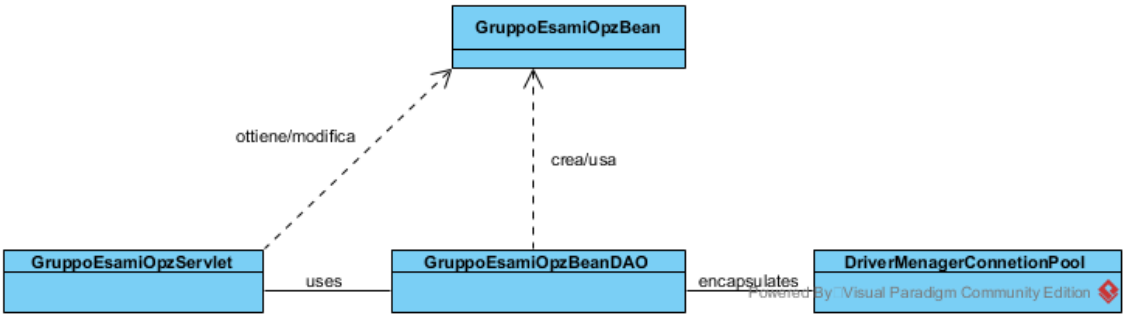

Figura 5 Pattern DAO per gruppi di esami opzionali

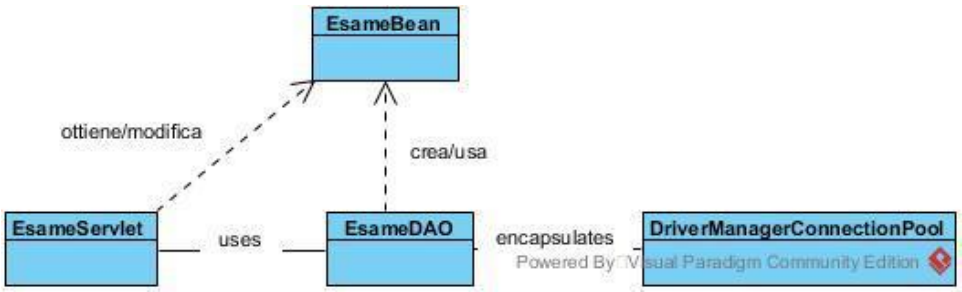

Figura 6 Pattern DAO per esami

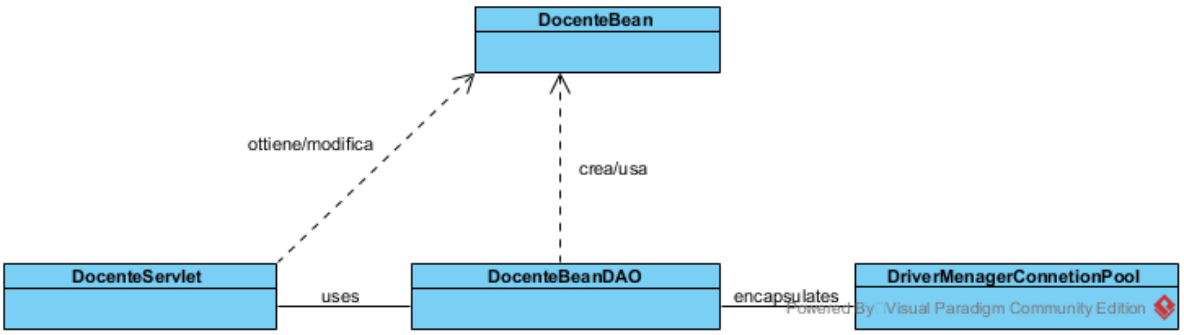

Figura 7 Pattern DAO per docenti

1.4.3. Singleton

Il pattern Singleton viene utilizzato quando si vuole garantire di avere un unico punto di accesso. Ad esempio, esso viene utilizzato quando si desidera avere un solo Window Manager oppure una sola Coda di Stampa oppure un unico accesso al database.
Un oggetto Singleton viene inizializzato nel momento in cui la classe viene invocata attraverso la definizione di un oggetto statico. La visibilità del suo costruttore viene modificata da public a private, così che non sia possibile istanziare la classe dall'esterno e fare in modo che soltanto la classe può istanziare sé stessa.

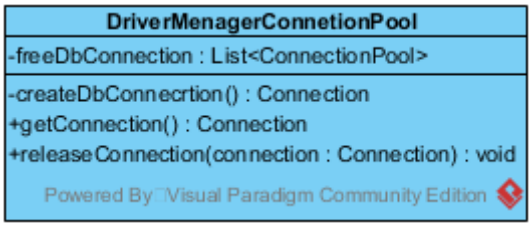

Figura 8 Classe di connessione del pattern Singleton

1.5. Definizioni, acronimi e abbreviazioni

All'interno del documento vengono utilizzati i seguenti acronimi o abbreviazioni:

- MVC: Model Control View.
- JSP: Java Server Pages.
- J2EE: Java Enterprise Edition 2.
- GUI: Graphic User Interface.
- HTML: Hyper Text Markup Language.
- DAO: Data Access Object.
- CSS: Cascading Style Sheets.
- CRUD: create, read, update, delete.

1.6. Riferimenti

Per la stesura di questo documento si è fatto riferimento al libro di testo *Object-Oriented Software Engineering* di Bernd Bruegge e Allen H. Dutoit.

2. Package

Di seguito, vengono illustrati i diagrammi dei package previsti dall'implementazione del sistema.

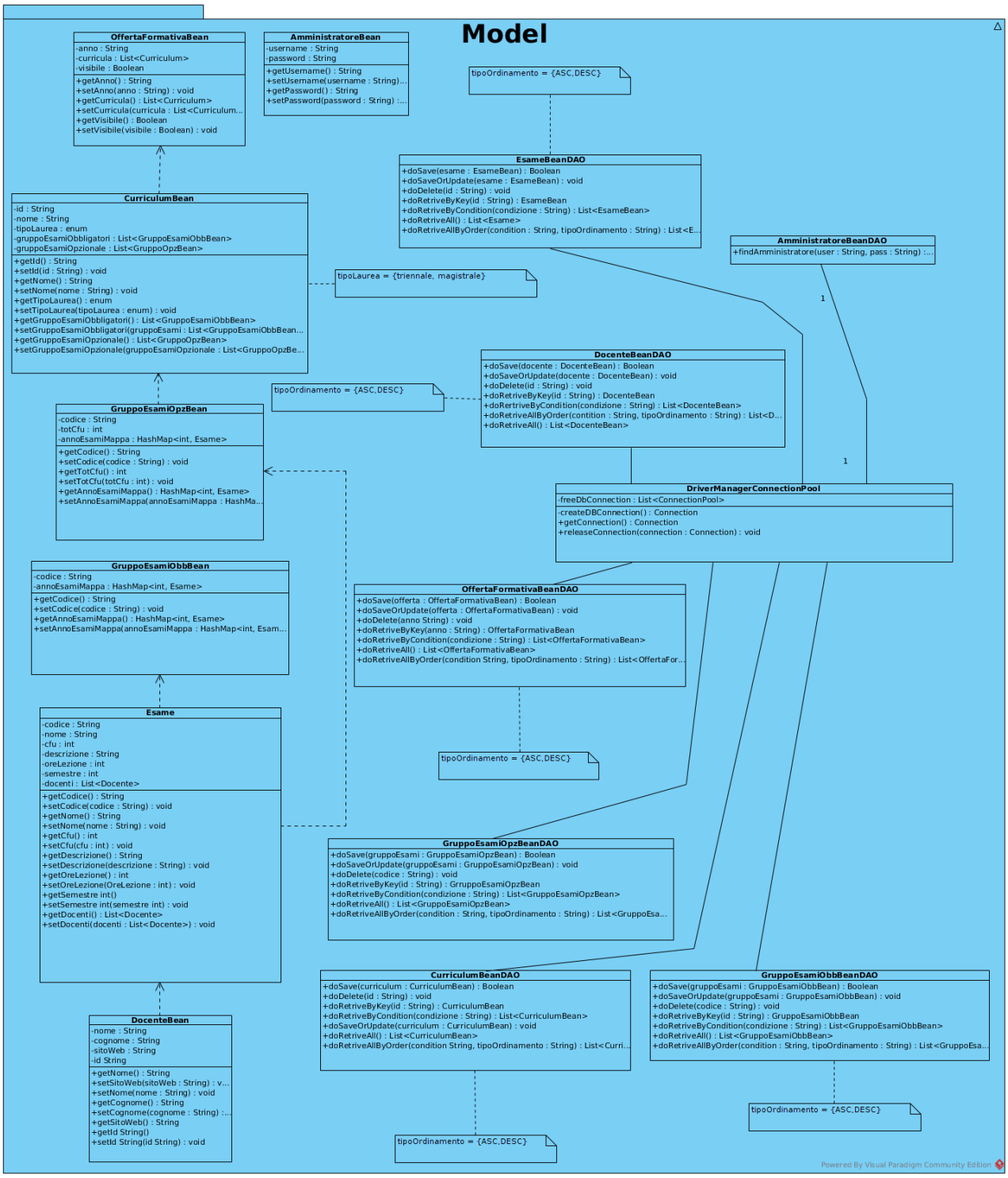

Figura 9 Package Model

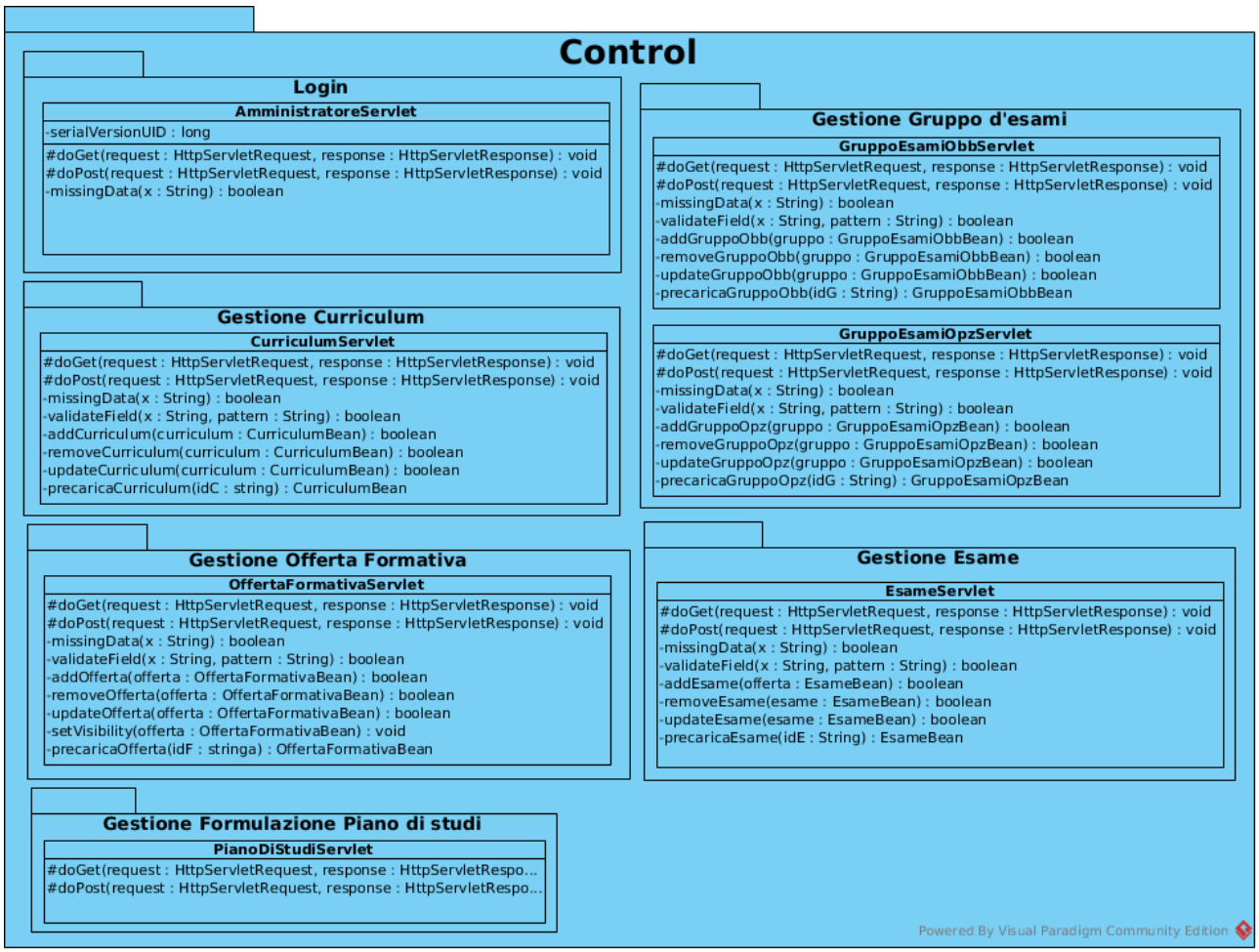

Figura 10 Package Control

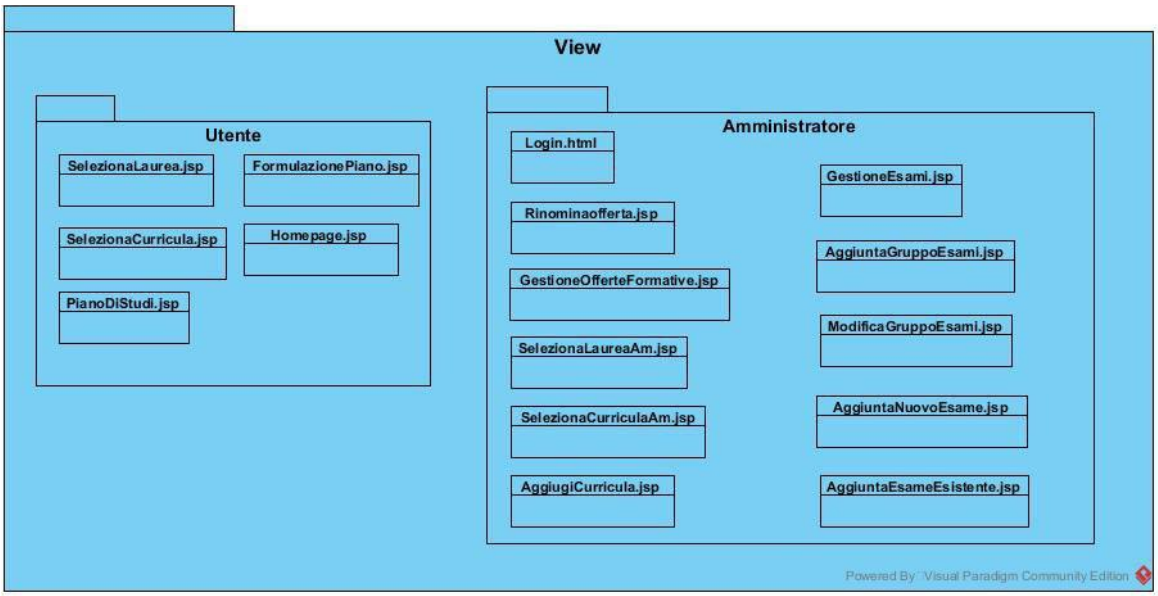

Figura 11 Package View

3. Interfacce delle classi

Di seguito, vengono elencate tutte le classi previste dal sistema, specificando le relative interfacce.

Nome classe	**AmministratoreServlet**
Descrizione	In fase di login si occupa di validare i dati di accesso all'area amministratore.
Pre-condizione	**context** AmministratoreServlet:: missingData(stringa); **pre:** stringa!=null && stringa!=""
Post-condizione	
Invarianti	

Nome classe	OffertaFormativaServlet
Descrizione	Si occupa di validare il nome dell'offerta formativa in fase di creazione. Inoltre, gestisce le funzionalità di aggiunta, modifica e rimozione dell'offerta formativa. Può anche cambiare la visibilità di un'offerta formativa.
Pre-condizione	**context** OffertaFormativaServlet::missingData(stringa); **pre:** stringa!=null && stringa!="" **context** OffertaFormativaServlet:: validateField(stringa, pattern); **pre:** stringa.matches(pattern) == true **context** OffertaFormativaServlet:: addOfferta(offertaFormativaBean); **pre:** offertaFormativaBean.exists() == false && offertaFormativaBean!= null **context** OffertaFormativaServlet:: removeOfferta(offertaFormativaBean); **pre:** offertaFormativaBean.exists() == true && offertaFormativaBean!= null **context** OffertaFormativaServlet::updateOfferta(offertaFormativaBean); **pre:** offertaFormativaBean.exists() == true && offertaFormativaBean!= null **context** OffertaFormativaServlet:: setVisibility(offertaFormativaBean); **pre:** offertaFormativaBean.exists() == true && offertaFormativaBean!= null
Post-condizione	
Invarianti	

Nome classe	CurriculumServlet
Descrizione	Si occupa di validare l'id, il nome, tipo di laurea, lista dei gruppi di esami dei curriculum in fase di creazione. Inoltre gestisce le funzionalità di aggiunta, modifica e rimozione dei curriculum.
Pre-condizione	**context** curriculumServlet:: missingData(stringa); **pre:** stringa!=null && stringa!="" **context** curriculumServlet:: validateField(stringa, pattern); **pre:** stringa.matches(pattern) == true **context** curriculumServlet:: addCurriculum(curriculumBean); **pre:** curriculumBean.exists() == false && curriculumBean!= null **context** curriculumServlet:: removeCurriculum(curriculumBean); **pre:** curriculumBean.exists() == true && curriculumBean!= null **context** curriculumServlet:: updateCurriculum(curriculumBean); **pre:** curriculumBean.exists() == true && curriculumBean!= null
Post-condizione	**Context** curriculumServlet::addCurriculum(curriculumBean); **post:** lista<gruppi di esami> == null
Invarianti	

Nome classe	**GruppiEsamiObbServlet**
Descrizione	Si occupa di validare il codice, anno dei curriculum e lista di esami in fase di creazione. Inoltre, gestisce le funzionalità di aggiunta, modifica e rimozione dei gruppi di esami obbligatori.
Pre-condizione	**context** GruppiEsamiObbServlet:: missingData(stringa); **pre:** stringa!=null && stringa!="" **context** GruppiEsamiObbServlet:: validateField(stringa, pattern); **pre:** stringa.matches(pattern) == true **context** GruppiEsamiObbServlet:: addGruppoObb(GruppoEsamiObbBean); **pre:** GruppoEsamiObbBean.exists() == false && GruppoEsamiObbBean!= null **context** GruppiEsamiObbServlet:: removeGruppoObb(GruppoEsamiObbBean); **pre:** GruppoEsamiObbBean.exists() == true && GruppoEsamiObbBean!= null **context** GruppiEsamiObbServlet:: updateGruppoObb(GruppoEsamiObbBean); **pre:** GruppoEsamiObbBean.exists() == true && GruppoEsamiObbBean!= null
Post-condizione	**Context** GruppiEsamiObbServlet::addGruppoObb(GruppoEsamiObbBean); **post:** lista<esami> == null
Invarianti	

Nome classe	**GruppiEsamiOpzServlet**
Descrizione	Si occupa di validare il codice, anno dei curriculum e lista di esami in fase di creazione. Inoltre gestisce le funzionalità di aggiunta, modifica e rimozione dei gruppi di esami opzionali.

Pre-condizione	**context** GruppiEsamiOpzServlet:: missingData(stringa);
	pre: stringa!=null && stringa!=""
	
	context GruppiEsamiOpzServlet:: validateField(stringa, pattern);
	pre: stringa.matches(pattern) == true
	
	context GruppiEsamiOpzServlet::
	addGruppoOpz(GruppoEsamiOpzBean);
	pre: GruppoEsamiOpzBean.exists() == false &&
	GruppoEsamiOpzBean!= null
	
	context GruppiEsamiOpzServlet::
	removeGruppoOpz(GruppoEsamiOpzBean);
	pre: GruppoEsamiOpzBean.exists() == true &&
	GruppoEsamiOpzBean!= null
	
	context GruppiEsamiOpzServlet::
	updateGruppoOpz(GruppoEsamiOpzBean);
	pre: GruppoEsamiOpzBean.exists() == true &&
	GruppoEsamiOpzBean!= null

Post-condizione	**context** GruppiEsamiOpzServlet::addGruppoOp<(GruppoEsamiOpzBean); **post:** lista<esami> == null
Invarianti	

Nome classe	EsameServlet
Descrizione	Si occupa di validare il codice, nome, cfu, descrizione, ore lezione, semestre, lista di docenti fase di creazione. Inoltre, gestisce le funzionalità di aggiunta, modifica e rimozione di un esame.
Pre-condizione	**context** EsameServlet:: missingData(stringa); **pre:** stringa!=null && stringa!="" **context** EsameServlet:: validateField(stringa, pattern); **pre:** stringa.matches(pattern) == true **context** EsameServlet:: addEsame(EsameBean); **pre:** EsameBean.exists() == false && EsameBean!= null **context** EsameServlet:: removeEsame(EsameBean); **pre:** EsameBean.exists() == true && EsameBean!= null **context** EsameServlet:: updateEsame(EsameBean); **pre:** EsameBean.exists() == true && EsameBean!= null
Post-condizione	
Invarianti	

Nome classe	PianoDiStudiServlet
Descrizione	Si occupa di validare il codice, nome, cfu, descrizione, ore lezione, semestre, lista di docenti fase di creazione. Inoltre, gestisce le funzionalità di aggiunta, modifica e rimozione di un esame.
Pre-condizione	**context** EsameServlet:: missingData(stringa); **pre:** stringa!=null && stringa!="" **context** EsameServlet:: validateField(stringa, pattern); **pre:** stringa.matches(pattern) == true **context** EsameServlet:: addEsame(EsameBean); **pre:** EsameBean.exists() == false && EsameBean!= null **context** EsameServlet:: removeEsame(EsameBean); **pre:** EsameBean.exists() == true && EsameBean!= null **context** EsameServlet:: updateEsame(EsameBean); **pre:** EsameBean.exists() == true && EsameBean!= null
Post-condizione	
Invarianti	