ODD
Object Design Document

BeYourChoice

Riferimento	BYC_RAD_V_1.5, BYC_SDD_V_1.2
Versione	1.1
Data	15/12/2024
Destinatario	Prof.ssa Filomena Ferrucci, Prof. Fabio Palomba
Presentato da	Aldo Damiano, Augusto Persico, Federica Iuliano, Fabrizio Cozzolino
Approvato da	Marco Ciano, Giuseppe D'Avino

Revision History

Data	Versione	Descrizione	Autori
18/11/2024	1.0	Prima stesura del documento	Aldo Damiano Federica Iuliano Augusto Persico Fabrizio Cozzolino
19/11/2024	1.1	Revisione dell'Introduzione, dei package e delle class interfaces	Aldo Damiano Federica Iuliano Augusto Persico Fabrizio Cozzolino

Sommario

1 Introduzione **4**
1.1 Object Design Trade-Off 4
1.2 Linee Guida per la Documentazione delle Interfacce 4
1.2.1 Package 4
1.2.2 Classi 5
1.2.3 Metodi 5
1.2.4 Nomi costanti 5
1.2.5 Nomi dei parametri 5
1.2.6 Nomi delle variabili locali 5
1.2.7 Enum names 5
1.2.8 Script Python 5
1.2.9 Pagine HTML 6
1.2.10 Fogli di stile CSS 6
1.2.11 Database MongoDB 6
1.2.12 Rotte Flask 6
1.3 Definizioni, acronimi e Abbreviazioni 6
1.4 Riferimenti 7
2 Design Pattern **8**
2.1 Facade Pattern 8
2.2 Observer Pattern 9
3 Package **11**
3.1 View 11
3.2 Model 12
3.3 Controller 13
4 Class Interfaces **16**
5 Glossario **27**

1 Introduzione

1.1 Object Design Trade-Off

Durante la fase di Object Design per il sistema BeYourChoice, sono stati individuati i seguenti trade-off per garantire il bilanciamento tra requisiti funzionali e non funzionali:

Response Time vs. Security: Il sistema darà priorità alla sicurezza rispetto al tempo di risposta, implementando controlli aggiuntivi per proteggere i dati degli utenti, anche se ciò potrà rallentare leggermente l'esperienza.

Robustness vs. Usability: La robustezza sarà prioritaria rispetto all'usabilità, implementando controlli stringenti per validare gli input degli utenti e prevenire errori o comportamenti imprevisti, anche a scapito di una maggiore complessità dell'interfaccia.

Reliability vs. Changeability: L'affidabilità sarà preferita alla modificabilità, adottando un approccio che limita i cambiamenti frequenti per garantire stabilità e continuità operativa, riducendo il rischio di regressioni o comportamenti inattesi.

Availability vs. Portability: Il sistema darà priorità alla disponibilità, assicurando che l'applicazione web sia sempre accessibile tramite browser sui PC degli utenti. Tuttavia, questa scelta riduce la portabilità, poiché il sistema non sarà ottimizzato per l'uso su dispositivi diversi dai PC, come smartphone o tablet.

Extensibility vs. Response Time: Il sistema privilegerà l'estendibilità rispetto al tempo di risposta, adottando una progettazione modulare e scalabile che facilita l'aggiunta di nuove funzionalità, anche se ciò può comportare un lieve impatto sulle prestazioni in tempo reale.

1.2 Linee Guida per la Documentazione delle Interfacce

Gli sviluppatori dovranno attenersi a precise linee guida per la stesura del codice, con l'obiettivo di garantire leggibilità, uniformità e manutenibilità. Per lo sviluppo in Python, utilizzato per la logica di business, e Flask, impiegato come framework web, le convenzioni adottate seguono gli standard consolidati di Python, e le best practices di Flask, con un'attenzione particolare all'architettura MVC e alle specificità del progetto BeYourChoice.

1.2.1 Package

I nomi dei package devono essere scritti in **lower_snake_case,** un formato di scrittura che utilizza solo lettere minuscole e separa le parole con un underscore "_". Inoltre, i package devono essere organizzati per rispecchiare l'architettura MVC e una struttura modulare.

1.2.2 Classi

- Le classi devono seguire la convezione upperCamelCase.
- Le classi devono rappresentare entità significative del dominio o componenti logici del sistema.

1.2.3 Metodi

- I nomi dei metodi devono essere scritti in lower_snake_case.

1.2.4 Nomi costanti

- Le costanti devono essere scritte in UPPER_CASE_WITH_UNDERSCORES.
- Devono rappresentare valori immutabili e definiti a livello globale o nel contesto locale.

1.2.5 Nomi dei parametri

I nomi dei parametri devono seguire le seguenti convenzioni:

- I nomi dei parametri devono essere scritti in lower_snake_case.
- Non è consentito l'uso di parametri con un singolo carattere, tranne nei casi richiesti da librerie esterne o framework.

1.2.6 Nomi delle variabili locali

Le variabili locali devono seguire la convenzione lower_snake_case e avere nomi descrittivi.

1.2.7 Enum names

I nomi degli enum devono seguire la convenzione upperCamelCase.

1.2.8 Script Python

Gli script python devono rispettare le seguenti convenzioni:

- Gli script devono avere uno scopo chiaro e non devono mescolare logiche diverse.
- Gli script devono essere in lowerCamelCase.
- Ogni script deve iniziare con un'intestazione commentata che descriva il file.
- Le funzioni e le classi devono essere documentate con docString.

1.2.9 Pagine HTML

Le pagine HTML devono rispettare lo standard HTML5.
I file HTML devono seguire le seguenti convenzioni:

- I nomi dei file devono essere in lowerCamelCase.
- I tag devono essere correttamente indentati.
- Ogni file deve essere organizzato per conteso d'uso (es. studenteDashboard.html).

1.2.10 Fogli di stile CSS

I fogli di stile (CSS) devono seguire le seguenti convenzioni:

- Tutti gli stili non in-line devono essere collocati in fogli di stile separati;
- Ogni foglio di stile deve essere iniziato da un commento descrittivo.
- I selettori e le proprietà devono essere chiaramente indentati.

1.2.11 Database MongoDB

I nomi delle collection devono seguire le seguenti regole:

- Devono essere scritti in upperCamelCase.
- Devono rappresentare un concetto chiaro del dominio (es. users, classes).

I nomi dei campi devono seguire le seguenti regole:

- Devono seguire la convenzione lower_snake_case.
- Devono essere descrittivi ed esplicativi.

1.2.12 Rotte Flask

Ogni rotta deve essere documentata tramite docstring e includere:

- Metodo HTTP.
- Endpoint.
- Parametri e risposta.

1.3 Definizioni, acronimi e Abbreviazioni

Acronimi:

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

Definizioni:

- **upperCamelCase**: Tecnica di naming delle variabili in cui la prima lettera di ogni parola è maiuscola.

- **lowerCamelCase**: Tecnica di naming in cui un nome contiene più parole unite insieme come una singola parola. La prima parola è sempre composta da lettere minuscole, inclusa la prima lettera. Le parole che seguono avranno solo la prima lettera maiuscola.
- **lower_snake_case** è una convenzione di naming in cui tutte le lettere sono minuscole e le parole sono separate da underscore (_).
- **Upper_case_underscore** è una convenzione di naming utilizzata principalmente per identificatori di costanti in molti linguaggi di programmazione. In questa convenzione:
  - Tutte le lettere sono maiuscole.
  - Le parole sono separate da underscore (_), migliorando la leggibilità.
- **HTML:** Linguaggio di programmazione utilizzato per lo sviluppo di pagine Web.
- **CSS:** Acronimo di Cascading Style Sheets, è un linguaggio usato per definire la formattazione delle pagine Web.
- **Python:** Linguaggio di programmazione ad alto livello, orientato a oggetti, adatto a sviluppare applicazioni distribuite, scripting, computazione numerica e system testing.
- **MongoDB:** DBMS non relazionale, orientato ai documenti.
- **MVC:** Acronimo di Model-View-Controller, è un pattern architetturale molto diffuso nello sviluppo di sistemi software.

1.4 Riferimenti

Requirements Analysis Document (BYC_RAD_V_1.5).

System Design Document (BYC_SDD_V_1.2).

Slide del corso, presenti sulla piattaforma e-learning.

Libro: *Object Oriented Software Engineering (Using UML, Patterns and Java, Prentice Hall)*
Autori: *B. Bruegge* & *A.H. Dutoit.*

2 Design Pattern

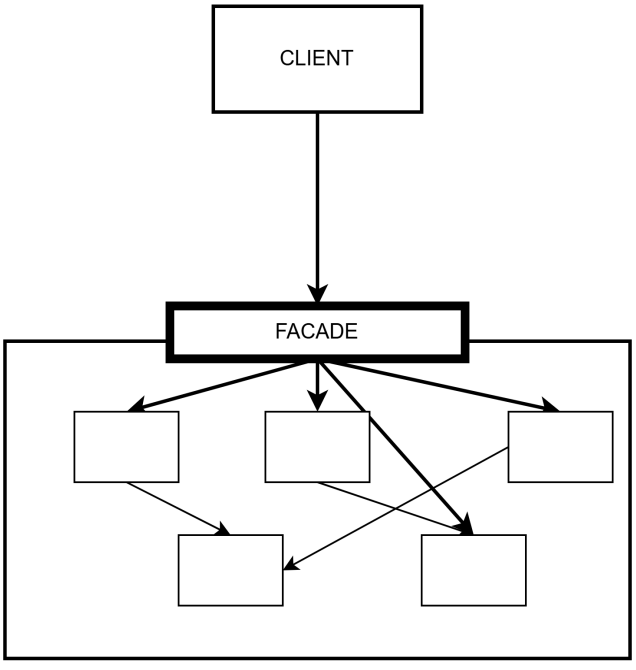
2.1 Facade Pattern

Nome e classificazione: Il Facade Pattern è un pattern strutturale che fornisce una classe di interfaccia semplificata a un sistema complesso. Fa parte dei Structural Pattern.

Scopo: Il Facade Pattern fornisce un'interfaccia semplificata e unificata di un sottosistema complesso. Fornisce un punto di accesso unico ad esso e aiuta a ridurre le dipendenze tra i componenti del sistema, migliorando la leggibilità e la manutenibilità del codice.

Applicabilità: Fornisce un'interfaccia semplificata permettendo ai client di interagire con un set di funzionalità senza conoscere i dettagli interni. Viene utilizzato per ridurre le dipendenze tra il client e le classi di un sottosistema. Inoltre, viene utilizzato per centralizzare le operazioni su più componenti, semplificando l'accesso e migliorando la coesione del codice. È particolarmente utile in contesti in cui un sistema complesso deve essere accessibile in modo semplice, come un sistema di gestione delle notifiche in cui diverse sottosistemi interagiscono, ma il client deve interagire solo con un'unica interfaccia.

Struttura:



Partecipanti: il Facade Pattern utilizza tre attori:

1. **Facade:** ovvero la classe che fornisce un'interfaccia semplificata (punto di accesso unificato) al sistema complesso.

2. **Classi del sottosistema:** ovvero le classi che contengono la logica effettiva e le funzionalità specifiche del sistema.
3. **Client:** ovvero il codice che interagisce con il Facade. Invece di comunicare con le varie classi del sottosistema, il client utilizza il Facade per eseguire operazioni, beneficiando di un'interfaccia semplificata.

Conseguenze: Offre un'interfaccia unificata che semplifica l'accesso a un sottosistema complesso, nascondendone i dettagli interni. In questo modo, i client interagiscono solo con il Facade senza conoscere i componenti sottostanti, garantendo un accoppiamento debole tra client e sottosistema. Questo riduce la dipendenza dalle classi interne e rende il sistema più facile da usare e manutenere.

Utilizzo: nel nostro sistema software verrà utilizzato per gestire il sistema di chat.

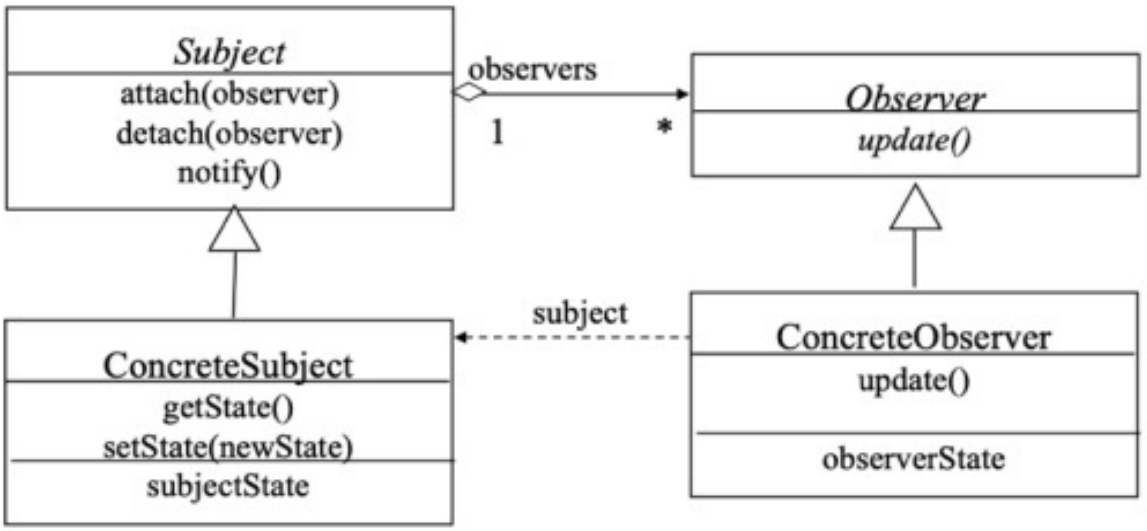
2.2 Observer Pattern

Nome e classificazione: Observer Pattern. Fa parte dei Behavioral Pattern.

Scopo: L'Observer Pattern si occupa di definire una relazione di dipendenza tra oggetti, in cui un cambiamento nello stato di un oggetto (il soggetto) porta automaticamente all'aggiornamento di tutti gli oggetti dipendenti (gli osservatori). Questo pattern è utile per gestire flussi di controllo complessi, dove diversi oggetti devono essere sincronizzati in tempo reale in risposta ai cambiamenti, senza che ci sia un accoppiamento diretto e rigido tra di loro.

Applicabilità: Definisce una dipendenza uno-a-molti tra gli oggetti, in cui quando uno di essi cambia stato, tutte le sue dipendenze vengono automaticamente notificate e aggiornate. Questo pattern è utilizzato per garantire la coerenza tra oggetti che condividono stati ridondanti e per ottimizzare la gestione delle modifiche in batch, mantenendo sincronizzati tutti gli stati correlati. Questo pattern è utile in applicazioni in tempo reale, come il sistema di notifiche, dove molti utenti (osservatori) devono essere aggiornati ogni volta che una notifica (soggetto) cambia.

Struttura:



Partecipanti: L' Observer Pattern utilizza quattro attori:

4. **Subject (Soggetto):** È l'oggetto osservato, che mantiene lo stato e invia notifiche agli osservatori quando questo stato cambia.
5. **Observer (Osservatore):** È l'oggetto che si iscrive per ricevere notifiche dal soggetto. Ogni osservatore reagisce ai cambiamenti dello stato del soggetto in modo specifico.
6. **Concrete Subject (Soggetto Concreto):** È una classe che implementa il soggetto e gestisce la lista di osservatori, notificando loro quando il suo stato cambia.
7. **Concrete Observer (Osservatore Concreto):** È una classe che implementa l'interfaccia dell'osservatore e aggiorna il proprio stato in base alle modifiche nel soggetto.

Conseguenze: Garantisce che, quando un oggetto cambia stato, un numero illimitato di oggetti dipendenti vengano aggiornati automaticamente. Inoltre, consente a un oggetto di notificare un numero illimitato di altri oggetti. Tuttavia, l'implementazione di base dell'Observer Pattern può portare a problemi di gestione della memoria, come il "problema dell'ascoltatore scaduto", poiché richiede sia una registrazione esplicita che una cancellazione esplicita, simile al Dispose Pattern. Questo accade perché il soggetto mantiene riferimenti forti agli osservatori, impedendo che vengano liberati dalla memoria e rischiando quindi perdite di memoria.

Utilizzo: nel nostro sistema software verrà utilizzato per gestire il sistema di notifiche.

3 Package

3.1 View

Il package view è la parte dell'architettura software che gestisce la presentazione delle informazioni e l'interazione con l'utente. Si occupa di mostrare i dati elaborati dal sistema, oltre a raccogliere input dall'utente.
Il package view include script Python che reindirizzano alle pagine HTML dove si trovano nella cartella **templates**, che offrono contenuti e funzionalità per tutti i ruoli del sistema. Determinati elementi della view vengono costruiti in modo dinamico dal controller per poter soddisfare esigenze specifiche. Alcune funzionalità sono riservate a un tipo specifico di utente, mentre altre sono accessibili globalmente.
La struttura è organizzata come segue:

Generico

- registrazioneLogin.html: Consente la registrazione di nuovi utenti e l'accesso agli utenti già registrati.
- gestioneProfilo.html: Consente di poter visualizzare il proprio profilo e modificare i propri dati se necessario.
- error404.html: Visualizzazione di una pagina di errore per le funzionalità non implementate.

Docente

- creazioneCV.html: Creazione di una nuova classe virtuale.
- classeDocente.html: Visualizzazione della classe virtuale.
- inserimentoStudente.html: Consente di poter inserire uno studente all'interno della classe.
- dashboardDocente.html: Permette di monitorare classi e studenti.
- classificaClasse.html: Visualizzazione classifica della classe.
- storicoStudenti.html: Visualizzazione storico studenti.
- materialeDocente.html: Consente di poter visualizzare il materiale didattico.
- caricamentoMateriale.html: Interfaccia per caricare il materiale didattico.
- modificaMateriale.html: Consente di poter modificare il titolo e la descrizione del materiale didattico precedentemente caricato.
- scenarioVirtuale.html: Modulo per configurare scenari virtuali.
- visore.html: Modulo per simulare la connessione allo scenario virtuale.
- creaQuiz.html: Consente di creare un quiz e di poterne visualizzare l'anteprima.
- quizPrecedenti.html: Visualizzazione dei quiz precedentemente creati.
- domandeQuizPrecedenti.html: Visualizzazione delle domande del quiz precedentemente creato.

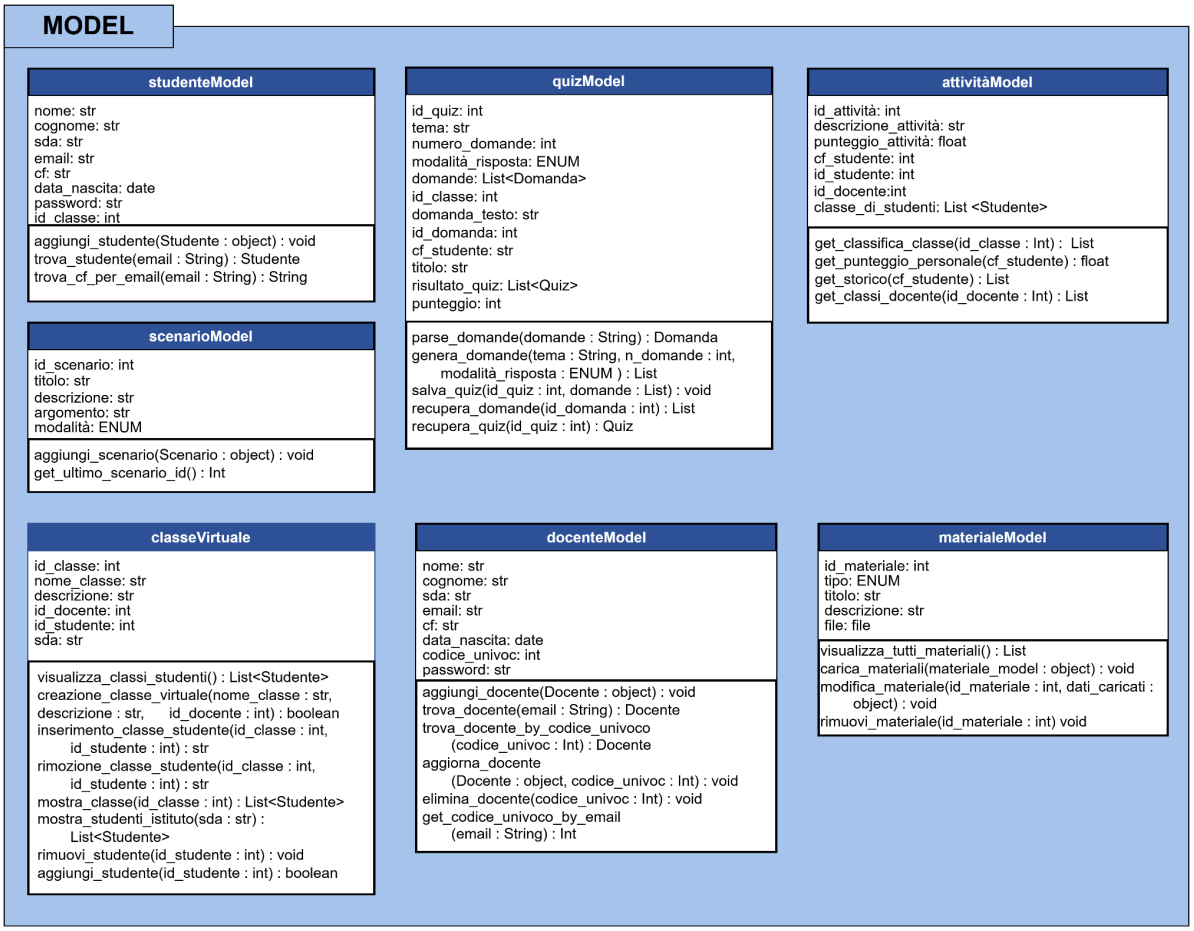
- risultatiQuizPrecedenti.html: Visualizzazione dei risultati del quiz precedentemente creato.

Studente

- classeStudente.html: Visualizzazione della classe virtuale.
- materialeStudente.html: Consente di poter visualizzare il materiale didattico.
- dashboardStudente.html: Permette di visualizzare i propri progressi personali.
- quizDisponibile.html: Visualizzazione del quiz reso disponibile dal docente.
- quiz.html: Accesso al quiz assegnato dal docente.
- noClasse.html: Visualizzazione di una pagina di errore che viene richiamata quando uno studente non appartiene ad una classe.

3.2 Model

Il package model è la componente del sistema responsabile della gestione dei dati e della loro rappresentazione logica. Si occupa di fornire l'accesso ai dati dell'applicazione, di mantenere lo stato del sistema e di definire la logica di business.
Le classi rappresentano le entità principali del sistema. Ogni classe inclusa nel pacchetto offre metodi che consentono di recuperare e gestire le informazioni necessarie al funzionamento dell'applicazione.
I moduli all'interno del package sono: Studente, Docente, Materiale Didattico, Classe Virtuale, Quiz, Scenario, Attività.

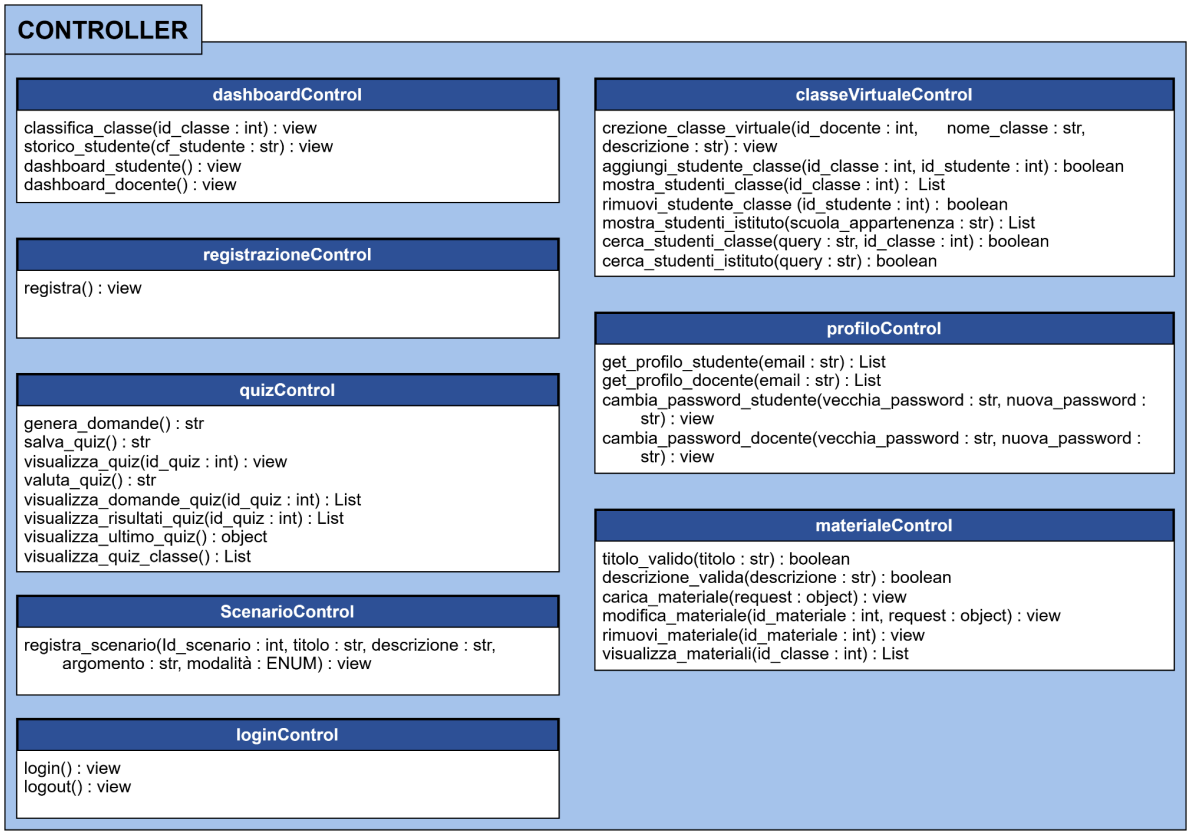


3.3 Controller

Il package controller si occupa di gestire la logica applicativa, coordinando l'interazione tra il model e la view. È responsabile dell'elaborazione delle richieste dell'utente, del controllo del flusso logico e dell'invio dei dati appropriati alla vista per la presentazione.
Esso è formato dai moduli:

- **registrazioneControl** (è responsabile della gestione del processo di registrazione, interfacciandosi direttamente con i model **studenteModel** e **docenteModel**, oltre che con la view registrazione.html).
- **loginControl** (è responsabile della gestione del processo di login e del logout, interfacciandosi direttamente con i model **studenteModel** e **docenteModel**, oltre che con la view registrazioneLogin.html).
- **profiloControl** (è responsabile della gestione del profilo dello studente, interfacciandosi direttamente con i model **studenteModel** e **docenteModel**, oltre che con la view gestioneProfilo.html).

- **classeVirtualeControl** (è responsabile della gestione della classe virtuale, interfacciandosi direttamente con il model **classeVirtualeModel**, oltre che con le view classeDocente.html, classeStudente.html, creazioneCV.html, e inserimentoStudente.html).
- **materialeControl** (è responsabile della gestione del materiale didattico, interfacciandosi direttamente con il model **materialeModel**, oltre che con le view materialeDocente.html, materialeStudente.html, caricamentoMateriale.html, e modificaMateriale.html).
- **scenarioControl** (è responsabile della gestione dello scenario virtuale, interfacciandosi direttamente con il model **scenarioModel**, oltre che con la view scenarioVirtuale.html).
- **quizControl** (è responsabile della gestione dei quiz, interfacciandosi direttamente con il model **quizModel**, oltre che con le view creaQuiz.html, quizPrecedenti.html, domandeQuizPrecedenti.html, risultatiQuizPrecedenti.html, quizDisponibile.html, quiz.html).
- **dashboardControl** (è responsabile della gestione della dashboard, interfacciandosi direttamente con il model **attivitaModel**, oltre che con le view dashboardDocente.html, dashboardStudente.html, classificaClasse.html e storicoStudenti.html).


CONTROLLER
dashboardControl
classifica_classe(id_classe : int) : view
storico_studente(cf_studente : str) : view
dashboard_studente() : view
dashboard_docente() : view
registrazioneControl
registra() : view
quizControl
genera_domande() : str
salva_quiz() : str
visualizza_quiz(id_quiz : int) : view
valuta_quiz() : str
visualizza_domande_quiz(id_quiz : int) : List
visualizza_risultati_quiz(id_quiz : int) : List
visualizza_ultimo_quiz() : object
visualizza_quiz_classe() : List
ScenarioControl
registra_scenario(Id_scenario : int, titolo : str, descrizione : str, argomento : str, modalità : ENUM) : view
loginControl
login() : view
logout() : view
classeVirtualeControl
crezione_classe_virtuale(id_docente : int, nome_classe : str, descrizione : str) : view
aggiungi_studente_classe(id_classe : int, id_studente : int) : boolean
mostra_studenti_classe(id_classe : int) : List
rimuovi_studente_classe (id_studente : int) : boolean
mostra_studenti_istituto(scuola_appartenenza : str) : List
cerca_studenti_classe(query : str, id_classe : int) : boolean
cerca_studenti_istituto(query : str) : boolean
profiloControl
get_profilo_studente(email : str) : List
get_profilo_docente(email : str) : List
cambia_password_studente(vecchia_password : str, nuova_password : str) : view
cambia_password_docente(vecchia_password : str, nuova_password : str) : view
materialeControl
titolo_valido(titolo : str) : boolean
descrizione_valida(descrizione : str) : boolean
carica_materiale(request : object) : view
modifica_materiale(id_materiale : int, request : object) : view
rimuovi_materiale(id_materiale : int) : view
visualizza_materiali(id_classe : int) : List

4 Class Interfaces

NomeClasse	registazioneControl
Descrizione	Questa classe gestisce la funzionalità di registrazione al sistema, distinguendo tra gli utenti docente e studente.
Pre-Condizione	**context:** registrazioneControl: registra();
Post-Condizione	**context:** registazioneControl: registra(); **post:** Collection<Utente> (size+1);
Invarianti	**context:** registazioneControl: registra(); **inv:** studente_dict != null; docente_dict != null;

NomeClasse	loginControl
Descrizione	Questa classe gestisce le funzionalità di autenticazione e disconnessione dal sistema.
Pre-Condizione	**context:** loginControl: login(); **context:** loginControl: logout();
Post-Condizione	**context:** loginControl: login(); **context:** loginControl: logout();
Invarianti	**context:** loginControl: login(); **inv:** email != null && password != null;

NomeClasse	classeVirtualeControl
Descrizione	Questa classe gestisce le funzionalità della classe virtuale.
Pre-Condizione	**context:** classeVirtualeControl crezione_classe_virtuale(Int id_docente, String nome_classe, String descrizione);

	context: classeVirtualeControl: aggiungi_studente_classe(Int id_studente, Int id_classe); **context:** classeVirtualeControl: mostra_studenti_classe(Int id_classe); **context:** classeVirtualeControl: rimuovi_studente_classe (Int id_studente); **context:** classeVirtualeControl: mostra_studenti_istituto(String sda); **context:** classeVirtualeControl: cerca_studenti_classe(String query, Int id_classe); **context:** classeVirtualeControl: cerca_studenti_istituto(String query);
Post-Condizione	**context:** classeVirtualeControl: crezione_classe_virtuale(Int id_docente, String nome_classe, String descrizione); **post:** collection<ClasseVirtuale>(size +1); **context:** classeVirtualeControl: aggiungi_studente_classe(Int id_studente, Int id_classe); **post:** collection<Studente>(size +1); **context:** classeVirtualeControl: mostra_studenti_classe(Int id_classe); **context:** classeVirtualeControl: rimuovi_studente_classe (Int id_studente); **post:** collection<Studente>(size -1);

	context: classeVirtualeControl:
	mostra_studenti_istituto(String sda);
	post: collection<Studente>();
	
	context: classeVirtualeControl:
	cerca_studenti_classe(String query, Int id_classe);
	post: collection<Studente>();
	
	context: classeVirtualeControl:
	cerca_studenti_istituto(String query);
	post: collection<Studente>();

Invarianti	**context:** classeVirtualeControl:
	crezione_classe_virtuale(Int id_docente, String nome_classe, String descrizione);
	inv: id_docente != null && nome_classe != null && descrizione != null;
	
	context: classeVirtualeControl:
	aggiungi_studente_classe(Int id_studente, Int id_classe);
	inv: id_studente != null && id_classe != null;
	
	context: classeVirtualeControl:
	mostra_studenti_classe(Int id_classe);
	inv: id_classe != null;
	
	context: classeVirtualeControl:
	rimuovi_studente_classe (Int id_studente);
	inv: id_studente != null;
	
	context: classeVirtualeControl:
	mostra_studenti_istituto(String sda);
	sda =! null;

	context: classeVirtualeControl: cerca_studenti_classe(String query, Int id_classe); **inv:** query != null; **context:** classeVirtualeControl: cerca_studenti_istituto(String query); **inv:** query != null;

NomeClasse	materialeControl
Descrizione	Questa classe gestisce le funzionalità del materiale didattico.
Pre-Condizione	**context:** materialeControl: carica_materiale(Object request); **context:** materialeControl: visualizza_materiali(Int id_classe); **context:** materialeControl: titolo_valido(String titolo); **context:** materialeControl:descrizione_valida(String descrizione); **context:** materialeControl: modifica_materiale(Int id_materiale, Object request); **context:** materialeControl: rimuovi_materiale(Int id_materiale);
Post-Condizione	**context:** materialeControl: carica_materiale(Object request); **post:** collection<MaterialeDidattico>(size +1); **context:** materialeControl: visualizza_materiali(Int id_classe); **post:** collection<MaterialeDidattico>();

	context: materialeControl: titolo_valido(String titolo); **post:** Boolean = true; **context:** materialeControl: descrizione_valida **post:** Boolean = true; **context:** materialeControl: modifica_materiale(Int id_materiale, Object request); **post:** collection<MaterialeDidattico>(alter); **context:** materialeControl: rimuovi_materiale(Int id_materiale); **post:** collection<MaterialeDidattico>(size -1);
Invarianti	**context:** materialeControl: carica_materiale(Object request); **inv:** request != null; **context:** materialeControl: visualizza_materiali(Int id_classe); **inv:** id_classe != null; **context:** materialeControl: titolo_valido(String titolo); **inv:** titolo != null; **context:** materialeControl: descrizione_valida(String descrizione); **inv:** descrizione != null; **context:** materialeControl: modifica_materiale(Int id_materiale, Object request); **inv:** id_materiale != null && request != null;

	context: materialeControl: rimuovi_materiale(Int id_materiale); **inv:** id_materiale != null;

NomeClasse	scenarioControl
Descrizione	Questa classe gestisce le operazioni relative agli scenari virtuali.
Pre-Condizione	**context:** scenarioControl: registra_scenario(Int id_scenario, String titolo, String descrizione, String argomento, ENUM modalita);
Post-Condizione	**context:** scenarioControl: registra_scenario(Int id_scenario, String titolo, String descrizione, String argomento, ENUM modalita); **post:** collection<Scenario>(size +1);
Invarianti	**context:** scenarioControl: registra_scenario(Int id_scenario, String titolo, String descrizione, String argomento, ENUM modalita); **Inv:** id_scenario != null && titolo != null && descrizione != null && argomento != null && modalita != null;

NomeClasse	quizControl
Descrizione	Questa classe gestisce le operazioni relative ai Quiz.
Pre-Condizione	**context:** quizControl: genera_domande(); **context:** quizControl: salva_quiz(); **context:** quizControl: visualizza_quiz(Int id_quiz); **context:** quizControl: valuta_quiz(); **context:** quizControl: visualizza_domande_quiz(Int id_quiz); **context:** quizControl: visualizza_risultati_quiz(Int id_quiz);

	context: quizControl: visualizza_ultimo_quiz(); **context:** quizControl: visualizza_quiz_classe();
Post-Condizione	**context:** quizControl: genera_domande(); **Post:** collection<Domanda>(size+1); **context:** quizControl: salva_quiz(); **Post:** collection<Quiz>(size+1); **context:** quizControl: valuta_quiz(); **Post:** String risultato; **context:** quizControl: visualizza_quiz(Int id_quiz); **Post:** collection<Quiz>(); **context:** quizControl: visualizza_domande_quiz(Int id_quiz); **Post:** collection<Domanda>(); **context:** quizControl: visualizza_risultati_quiz(Int id_quiz); **Post:** collection<Risultato>(); **context:** quizControl: visualizza_ultimo_quiz(); **Post:** Object quiz; **context:** quizControl: visualizza_quiz_classe(); **Post:** collection<Quiz>();
Invarianti	**context:** quizControl: visualizza_quiz(Int id_quiz); **Inv:** id_quiz != null; **context:** quizControl: visualizza_domande_quiz(Int id_quiz); **Inv:** id_quiz != null; **context:** quizControl: visualizza_risultati_quiz(Int

	id_quiz); **Inv:** id_quiz != null;

NomeClasse	dashboardControl
Descrizione	Questa classe gestisce le funzionalità relative alla dashboard.
Pre-Condizione	**context:** dashboardControl: classifica_classe(Int id_classe); **context:** dashboardControl: storico_studente(String cf_ studente); **context:** dashboardControl: dashboard_studente(); **context:** dashboardControl: dashboard_docente();
Post-Condizione	**context:** dashboardControl: classifica_classe(Int id_classe); **Post:** collection<Classe>(); **context:** dashboardControl: storico_studente(String cf_ studente); **Post:** collection<Attività>(); **context:** dashboardControl: dashboard_studente(); **Post:** collection<Attività>(); **context:** dashboardControl: dashboard_docente(); **Post:** collection<Classe>();
Invarianti	**context:** dashboardControl: classifica_classe(Int id_classe); **inv:** id_classe != null;

	context: dashboardControl: storico_studente(String cf_ studente); **inv:** cf_studente != null;

NomeClasse	profiloControl
Descrizione	Questa classe gestisce le funzionalità relative al profilo.
Pre-Condizione	**context:** profiloControl: get_profilo_studente(String email); **context:** profiloControl: get_profilo_docente(String email); **context:** profiloControl: cambia_password_studente(String vecchia_password, String nuova_password); **context:** profiloControl: cambia_password_docente(String vecchia_password, String nuova_password);
Post-Condizione	**context:** profiloControl: get_profilo_studente(String email); **Post:** collection<Studente>(); **context:** profiloControl: get_profilo_docente(String email); **Post:** collection<Studente>(); **context:** profiloControl: cambia_password_studente(String vecchia_password, String nuova_password); **Post:** collection<Studente>(alter); **context:** profiloControl: cambia_password_docente(String

	vecchia_password, String nuova_password); **Post:** collection<Docente>(alter);
Invarianti	**context:** profiloControl: get_profilo_studente(String email); **inv:** email != null; **context:** profiloControl: get_profilo_docente(String email); **inv:** email != null; **context:** profiloControl: cambia_password_studente(String vecchia_password, String nuova_password); **inv:** vecchia_password != null && nuova_password != null; **context:** cambia_password_docente(String vecchia_password, String nuova_password); **inv:** vecchia_password != null && nuova_password != null;

5 Glossario

- **MongoDB**: è un DBMS non relazionale, orientato ai documenti. Classificato come un database di tipo NoSQL, MongoDB si allontana dalla struttura tradizionale basata su tabelle dei database relazionali a favore di documenti in stile JSON con schema dinamico.
- **DBMS (Database Management System)**: è un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database, per questo detto anche "gestore o motore del database", è ospitato su architettura hardware dedicata oppure su semplice computer.
- **NoSQL:** i database NoSQL sono appositamente realizzati per modelli di dati specifici e hanno schemi flessibili per creare applicazioni moderne. Si discostano dai Database di tipo relazionale, infatti l'espressione "NoSQL" fa riferimento al linguaggio SQL, che è il più comune linguaggio di interrogazione dei dati nei database relazionali, qui preso a simbolo dell'intero paradigma relazionale.
- **JSON (Javascript Object Notation)**: è un semplice formato per lo scambio di dati. È basato sul linguaggio JavaScript, ma ne è indipendente. Viene usato in AJAX come alternativa a XML/XSLT.
- **CamelCase:** la notazione "a cammello", o in inglese CamelCase, è la pratica nata durante gli anni 70 di scrivere parole composte o frasi unendo tutte le parole tra loro, ma lasciando le loro iniziali maiuscole. Si può distinguere in un lowerCamelCase, in cui la prima lettera della prima parola viene lasciata minuscola, o in upperCamelCase, in cui la prima lettera della prima parola è maiuscola.
- **HTML (HyperText Markup Language):** è un linguaggio di markup nato per la formattazione e impaginazione di documenti ipertestuali disponibili nel web 1.0, oggi è utilizzato principalmente per il disaccoppiamento della struttura logica di una pagina web**.**
- **CSS (Cascading Style Sheets):** è un linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML ad esempio nei siti web e relative pagine web.
- **Trade-off:** il Trade-off è una situazione che implica una scelta tra due possibilità, in cui la perdita di valore di una costituisce un aumento di valore in un'altra.
- **Design Pattern:** Un design pattern è una soluzione progettuale standardizzata per risolvere problemi ricorrenti nello sviluppo software, migliorando la struttura, la manutenibilità e la riutilizzabilità del codice. Si tratta di modelli astratti, suddivisi in categorie come creazionali, strutturali e comportamentali.
- **MVC:** (Model-View-Controller) è un pattern architetturale utilizzato per separare i vari componenti di un'applicazione, migliorando la manutenibilità e la testabilità del codice. Questo pattern è particolarmente popolare nello sviluppo di

applicazioni web e desktop, e facilita la gestione delle interazioni utente, dei dati e della logica di business.