Università degli Studi di Salerno
Corso di Ingegneria del Software

EduCat
Object Design Document
Versione 1.3

Data: 21/01/2026

Coordinatore del progetto:

Nome	Matricola
Isabella Dima	0512119719

Partecipanti:

Nome	Matricola
Isabella Dima	0512119719
Anna Chiara Esposito	0512119143

Scritto da:	Anna Chiara Esposito

Revision History

Data	Versione	Descrizione	Autore
21/12/2025	1.0	Object Design Document: sottosistema	Isabella Dima Anna Chiara Esposito
14/01/2026	1.1	Modifiche all'Overview dei pacchetti e aggiunto Glossario	Anna Chiara Esposito
15/01/2026	1.2	Modifiche alle Interfacce delle Classi, modifiche all'indice	Anna Chiara Esposito
21/01/2026	1.3	Completato e corretto le interfacce delle classi	Anna Chiara Esposito

Indice

1. Introduzione

1.1 Compromessi di progettazione

Trade-off	Descrizione
Utilizzo di memoria vs Tempo di risposta	Il sistema darà priorità a tempi di risposta più rapidi rispetto a un utilizzo minimo della memoria. A tal fine saranno utilizzati Beans per trasferire dati tra i layer e le operazioni di ricerca verranno delegate al livello di database, alleggerendo il livello di applicazione.
Performance vs Manutenibilità	Il sistema favorirà la manutenibilità a lungo termine rispetto alla performance. Questo sarà realizzato utilizzando dei DAO e separando la logica di business in classi apposite, migliorando così la modularità e testabilità.

1.2 Linee guida per la documentazione delle interfacce

Questa sezione definisce le linee guida di documentazione delle interfacce e convenzioni nel codice che verranno adottate nell'Object Design di EduCat.
Queste convenzioni hanno lo scopo di aiutare gli sviluppatori a progettare le interfacce in maniera consistente e facilitare lo sviluppo definendo linee guida di leggibilità e facilità nella comunicazione.
Le convenzioni non si evolveranno nel corso del progetto.

1.1.1. Convenzioni sui nomi

- Le classi sono chiamate con singoli nomi con convenzione PascalCase (es. Utente, Lezione).
- I metodi sono denominati con sintagmi verbali in camelCase (es. login(), prenotaLezione()) mentre i campi e i parametri con sintagmi nominali in camelCase (es. email).
- Costanti ed enumerazioni sono scritti in maiuscolo (es. STATO)

1.1.2. Design delle interfacce

- Le interfacce pubbliche rendono disponibili solo le operazioni strettamente necessarie all'utilizzatore dell'interfaccia.
- I dettagli implementativi sono nascosti alle classi client.

1.1.3. Eccezioni

- Gli errori sono gestiti tramite eccezioni controllate e non valori di ritorno (es. SlotOccupatoException).
- Le eccezioni contengono messaggi che chiarificano la natura dell'errore.

1.1.4. Condizioni di Boundary

- Input invalidi risulteranno in eccezioni.
- Valori nulli, identificatori invalidi e altre condizioni di boundary sono gestite in maniera esplicita.

1.3 Definizioni, acronimi e abbreviazioni

- **Studenti**: utenti che ricercano lezioni private per migliorare la propria preparazione scolastica o universitaria.
- **Genitori**: nel caso di studenti minorenni, i genitori agiscono come referenti per la prenotazione e il pagamento delle lezioni.
- **Tutor**: figure abilitate a offrire lezioni private. Possono registrarsi, gestire il proprio profilo, definire tariffe e disponibilità e ricevere feedback dagli studenti.
- **Package**: collezioni di classi raggruppate insieme.
- **MVC**: Model-View-Controller: pattern di architettura software che separa la logica di business, la presentazione e l'accesso ai dati persistenti per migliore manutenibilità e maggiore facilità nel debugging.

1.4 Riferimenti

- RAD_EduCat v1.5
- SDD_EduCat v1.3
- Bernd Bruegge & Allen Dutoit - Object-Oriented Software Engineering: Using UML, Patterns, AND Java

2. Pacchetti

Questa sezione descrive la decomposizione del sistema in pacchetti (packages) Java, seguendo l'architettura MVC (Model-View-Controller) e la suddivisione in livelli logici (Interface, Application, Storage) definita nel System Design.
La struttura dei pacchetti è organizzata per separare chiaramente la logica di presentazione, di business e di persistenza.

2.1 Overview dei Pacchetti

Il sistema utilizza il root package it.unisa.educat.

Pacchetto	Descrizione	Dipendenze
it.unisa.educat.model	Contiene le classi entità (Java Beans) che rappresentano i dati del dominio (es. Utente, Lezione). Corrisponde ai dati definiti nel DataBase.	Nessuna (Data Transfer Objects).
it.unisa.educat.dao	Contiene le interfacce e le classi DAO (Data Access Object) per l'interazione con il database	it.unisa.educat.model

	MySQL tramite JDBC.	
it.unisa.educat.controller	Contiene le Servlet che gestiscono le richieste HTTP, che implementano la logica di business, i servizi dei sottosistemi e selezionano la vista di risposta.	It.unisa.educat.model, It.unisa.educat.dao
webapp	Contiene le pagine JSP (JavaServer Pages) e risorse statiche (HTML/CSS) per l'interfaccia utente.	Utilizza oggetti di it.unisa.educat.model ricevuti dal controller

2.2 Organizzazione dei File

La struttura delle directory del codice sorgente riflette la gerarchia dei pacchetti:

- **src/main/java/it/unisa/educat/model/** (Contiene le Data Transfer Objects/Bean come UtenteDTO.java, LezioneDTO.java, PrenotazioneDTO.java, SegnalazioneDTO.java)
- **src/main/java/it/unisa/educat/dao/** (Contiene le classi per l'accesso ai dati come GestioneUtenzaDAO.java, GestioneLezioneDAO.java e DatasourceManager.java)
- **src/main/java/it/unisa/educat/control/** (Contiene i sotto-pacchetti per sottosistema, ad esempio:
  - gestioneUtenza (es. LoginServlet.java, RegistrazioneServlet.java)
  - gestioneLezione (es. PrenotaLezioneServlet.java, CercaLezioneServlet.java)
  - gestioneSegnalazione (es. InviaSegnalazioneServlet.java))
- **src/main/webapp/** (Contiene le pagine JSP come homePageGenerica.jsp, login.jsp, i fogli di stile in /styles e le risorse statiche in /images)

3. Interfacce delle classi

Questa sezione descrive le interfacce pubbliche delle classi principali raggruppate per sottosistema. Per ogni sottosistema vengono dettagliate le classi Entità (Model), le classi di persistenza (DAO) e le classi di controllo (Controller). Per le operazioni che modificano lo stato del sistema, sono specificati i contratti (precondizioni e postcondizioni) in linguaggio OCL.

3.1 Gestione Utenza

Questo sottosistema gestisce la registrazione, l'autenticazione e la gestione del profilo degli utenti.

- **Classe UtenteDTO (Model)**
  - Descrizione: Data Transfer Object (DTO) che rappresenta i dati di un utente registrato nel sistema. Mappa le informazioni della tabella Utente del database e gestisce i dati per tutti i ruoli (Studente, Genitore, Tutor, Admin).
  - Invarianti: context UtenteDTO inv: self.email <> null AND self.email <> "" AND self.password <> null AND self.UID ≥ 0

- Attributi:
  - -UID: int
  - -nome: String
  - -cognome: String
  - -email: String
  - -password: String (Hashed)
  - -dataNascita: String
  - -via: String
  - -civico: String
  - -città: String
  - -CAP: String
  - -tipo: TipoUtente, enum("STUDENTE", "GENITORE", "TUTOR", "ADMIN")
  - -nomeFiglio: String
  - -cognomeFiglio: String
  - -dataNascitaFiglio: String
- Operazioni: Getter e Setter per tutti gli attributi.

- **Classe GestioneUtenzaDAO (Persistence)**
  - Descrizione**:** Classe che implementa il pattern Data Access Object (DAO) per gestire la persistenza dell'entità Utente. Si occupa di eseguire le operazioni CRUD (Create, Read, Update, Delete) verso il database relazionale, gestendo le connessioni JDBC e mappando i ResultSet SQL negli oggetti UtenteDTO.
  - Operazioni:

Operazione	Contratto(OCL)
+doSave(UtenteDTO utente) → boolean	**Descrizione**: Memorizza un nuovo utente nel database. Gestisce automaticamente l'inserimento dei dati aggiuntivi se l'utente è di tipo GENITORE (nome/cognome figlio). **Precondizioni**: Context GestioneUtenzaDAO:: doSave(UtenteDTO utente) → boolean pre: utente <> null AND self.utenti → forAll(u | u.email <> utente.email) **Postcondizioni**: Context GestioneUtenzaDAO:: doSave(UtenteDTO utente) → boolean post: result = true implies(self.utenti → includes(utente) AND self.utenti → size() = self.utenti@pre → size() + 1)
+doRetrieveByEmail(String email) → UtenteDTO	**Descrizione**: Ricerca un utente tramite l'indirizzo email. Usato principalmente per la fase di Login. Restituisce null se non trovato.

	Precondizioni:
	Context GestioneUtenzaDAO::
	doRetrieveByEmail(String email) → UtenteDTO
	pre: email <> null
	
	Postcondizioni:
	Context GestioneUtenzaDAO::
	doRetrieveByEmail(String email) → UtenteDTO
	post: if self.utenti → exists(u | u.email = email) then
	result = self.utenti → select(u | u.email = email) → first()
	else
	result = null
	endif

+doRetrieveAll() → List<UtenteDTO>	**Descrizione**: Restituisce una lista contenente tutti gli utenti registrati nel sistema. **Postcondizioni:** Context GestioneUtenzaDAO:: doRetrieveAll() → List<UtenteDTO> post: result→ forAll(u | self.utenti → includes(u)) AND result → size() = self.utenti → size()
+doDelete(int idUtente) → boolean	**Descrizione**: Rimuove permanentemente un utente dal database dato il suo identificativo univoco (UID). **Precondizione**: Context GestioneUtenzaDAO:: doDelete(int idUtente) → boolean pre: self.utenti → exists(u | u.UID = idUtente) **Postcondizione:** Context GestioneUtenzaDAO:: doDelete(int idUtente) → boolean post: result = true implies self.utenti → forAll(u | u.UID <> idUtente)

- **Classe GestioneUtenzaController (Control)**
  - Descrizione**:** Componente logico che raggruppa le Servlet del package it.unisa.educat.controller.gestioneutenza. Gestisce il flusso delle richieste HTTP relative agli utenti, gestendo l'interazione tra le View (JSP) e il Model (DAO). Si

occupa della validazione degli input, della gestione della sessione e della sicurezza (hashing password).

- Operazioni:

Operazione	Servlet Implentatativa	Contratto (OCL)

+login(String email, String password)	LoginServlet (doPost)	**Descrizione**:
		Verifica le credenziali, calcola l'hash della password e crea la sessione utente. Reindirizza alla home page specifica per ruolo.
		
		Precondizioni:
		Context GestioneUtenzaController::
		login(String email, String password)
		pre: email <> null AND email <> ""
		AND password <> null AND password
		<> ""
		
		Postcondizioni:
		Context GestioneUtenzaController::
		login(String email, String password)
		post: let targetUser =
		GestioneUtenzaDAO.utenti → select(u |
		u.email = email) → first() in
		if (targetUser <> null AND
		targetUser.password = hash(password))
		then session.getAttribute("utente") =
		targetUser AND session.isNew = false
		else
		session.getAttribute("utente") = null
		endif

+registraUtente(UtenteDTO utente)	RegistrazioneServlet (doPost)	**Descrizione**: Riceve i dati dal form, valida l'unicità dell'email e persiste il nuovo utente nel DB tramite il DAO. **Precondizioni:** context GestioneUtenzaController:: registraUtente(UtenteDTO utente) pre: utente.email <> null AND utente.password <> null AND not GestioneUtenzaDAO.utenti → exists(u | u.email = utente.email) **Postcondizioni:** context GestioneUtentenzaController:: registraUtente(UtenteDTO utente) post: GestioneUtenzaDAO.utenti →

		includes(utente) AND GestioneUtenzaDAO.utenti → size() = GestioneUtenzaDAO.utenti@pre → size() + 1
+logout()	LogoutServlet (doGet)	**Descrizione:** Invalida la sessione corrente (session.invalidate()) e reindirizza alla pagina pubblica. **Precondizioni:** Context GestioneUtenzaController::logout() pre: session.getAttribute("utente") <> null **Postcondizioni:** Context GestioneUtenzaController::logout() post: session.isInvalid() = true AND session.getAttribute("utente") = null
+eliminaAccount(int idUtente, UtenteDTO richiedente)	EliminaAccountServlet (doPost)	**Descrizione:** Permette a un utente di eliminare il proprio profilo o a un Admin di bannare un utente. Gestisce la chiusura della sessione se l'eliminazione è auto-indotta. **Precondizioni:** Context GestioneUtenzaController::eliminaAccount(int idUtente, UtenteDTO richiedente) pre: richiedente <> null AND GestioneUtenzaDAO.utenti → exists(u | u.UID = idUtente) AND (richiedente.UID = idUtente OR richiedente.tipo = TipoUtente::AMMINISTRATORE) **Postcondizioni:** Context GestioneUtenzaController::eliminaAccount(int idUtente, UtenteDTO richiedente) post: !GestioneUtenzaDAO.utenti → exists(u | u.UID = idUtente) AND (richiedente.UID = idUtente implies session.isInvalid())
+getListaUtenti(UtenteDTO richiedente, Map filtri) → List<UtenteDTO>	ListaUtentiServlet (doGet)	**Descrizione:** (Admin) Recupera la lista degli utenti registrati, applicando eventuali filtri di

		ricerca (nome, email, ruolo). **Precondizioni:** Context GestioneUtenzaController:: getListaUtenti(UtenteDTO richiedente, Map filtri) → List<UtenteDTO> pre: richiedente <> null AND richiedente.tipo = TipoUtente::AMMINISTRATORE **Postcondizioni:** Context GestioneUtenzaController:: getListaUtenti(UtenteDTO richiedente, Map filtri) → List<UtenteDTO> Post: result = GestioneUtenzaDAO.utenti → select(u | (filtri.email <> null implies u.email.includes(filtri.email)) AND (filtri.nome <> null implies u.nome.includes(filtri.nome)) AND (filtri.tipo <> null implies u.tipo = filtri.tipo)

3.2 Gestione Lezione

Questo sottosistema gestisce la pubblicazione degli annunci, la ricerca e la prenotazione delle lezioni.

- **Classe LezioneDTO (Model)**
  - Descrizione**:** Rappresenta una lezione offerta da un Tutor o prenotata da uno Studente. Contiene i dettagli temporali, economici e lo stato corrente dell'attività.
  - Invarianti: context LezioneDTO inv: self.prezzo ≥ 0 AND self.durata > 0 AND self.dataInizio < self.dataFine AND self.tutor <> null
  - Attributi:
    - -idLezione: int
    - -materia: String
    - -dataInizio: LocalDateTime
    - -dataFine: LocalDateTime
    - -durata: float
    - -prezzo: float
    - -modalitaLezione: ModalitaLezione, Enum(“ONLINE”, “PRESENZA”)
    - -tutor: UtenteDTO
    - -città: String
    - -stato: StatoLezione, Enum(“PIANIFICATA”, “PRENOTATA”, “CONCLUSA”, “ANNULLATA”)
  - Operazioni: Getter e Setter per tutti gli attributi.

- **Class Prenotazione (Model)**

- Descrizione**:** Rappresenta l'associazione tra uno Studente e una Lezione, includendo i dettagli del pagamento.
- Invarianti: context PrenotazioneDTO inv: self.importoPagato ≥ 0 AND self.studente <> null AND self.lezione <> null
- Attributi:
  - -idPrenotazione: int
  - -idTutor: int
  - -dataPrenotazione: LocalDate
  - -stato: StatoPrenotazione, enum (“ATTIVA”, “ANNULLATA”, “CONCLUSA”)
  - -importoPagato: float
  - -studente: UtenteDTO
  - -lezione: LezioneDTO
  - -numeroCarta: String
  - -intestatario: String
  - -cvv: int
  - -dataScadenza: String
  - -indirizzoFatturazione: String
- Operazioni: Getter e Setter per tutti gli attributi.

- **Classe GestioneLezioneDAO (Persistence)**
  - Operazioni:

Operazione	Contratto (OCL)
+doSaveLezione(LezioneDTO lezione) → boolean	**Descrizione:** Salva una nuova lezione offerta da un Tutor. Verifica preliminare che il tutor non abbia già impegni in quella fascia oraria. **Precondizioni:** context GestioneLezioneDAO:: doSaveLezione(LezioneDTO lezione) → boolean pre: lezione <> null AND !self.hasTutorLezioneInFasciaOraria(lezione.tutor.UID, lezione.dataInizio, lezione.dataFine) **Postcondizioni:** context GestioneLezioneDAO:: doSaveLezione(LezioneDTO lezione) → boolean post: result = true implies (self.lezioni → includes(lezione) AND lezione.stato = StatoLezione::PIANIFICATA)
+hasTutorLezioneInFasciaOraria(int idTutor, DateTime inizio, DateTime fine) → boolean	**Descrizione:** Verifica se un tutor ha già lezioni (pianificate o prenotate) che si sovrappongono all'intervallo temporale indicato. Restituisce true in caso di sovrapposizione. **Precondizioni:** context GestioneLezioneDAO::

	hasTutorLezioneInFasciaOraria(int idTutor, DateTime inizio, DateTime fine) → boolean pre: idTutor > 0 and inizio < fine **Postcondizioni:** context GestioneLezioneDAO:: hasTutorLezioneInFasciaOraria(int idTutor, DateTime inizio, DateTime fine) → boolean post: result = self.lezioni → exists(l | l.tutor.UID = idTutor AND (l.stato = StatoLezione::PIANIFICATA OR l.stato = StatoLezione::PRENOTATA) AND l.dataInizio < fine AND l.dataFine > inizio)
+doRetrieveByCriteria(CriteriRicerca criteri) → List<LezioneDTO>	**Descrizione**: Esegue una ricerca avanzata delle lezioni disponibili filtrando per materia, città, prezzo massimo, modalità. **Precondizioni:** context GestioneLezioneDAO:: doRetrieveByCriteria(CriteriRicerca criteri) → List<LezioneDTO> pre: criteri <> null **Postcondizioni:** context GestioneLezioneDAO:: doRetrieveByCriteria(CriteriRicerca criteri) → List<LezioneDTO> post: result → forAll(l | l.stato = StatoLezione::PIANIFICATA AND (criteri.materia <> null implies l.materia = criteri.materia) AND (criteri.citta <> null implies l.citta = criteri.citta) AND (criteri.prezzoMax > 0 implies l.prezzo ≤ criteri.prezzoMax) AND (criteri.modalita <> null implies l.modalitaLezione = criteri.modalita))
+prenotaLezione(PrenotazioneDTO prenotazione) → boolean	**Descrizione:** Gestisce la prenotazione di una lezione in modalità transazionale: salva la prenotazione e aggiorna lo stato della lezione a PRENOTATA. **Precondizioni:** context GestioneLezioneDAO:: prenotaLezione(PrenotazioneDTO prenotazione) → boolean pre: let lezioneTarget = self.getLezioneById(prenotazione.lezione.idLezione) in lezioneTarget.stato = StatoLezione::PIANIFICATA AND !self.hasStudentePrenotazioneInFasciaOraria(prenotazione.studente.UID, lezioneTarget.dataInizio, lezioneTarget.dataFine) AND prenotazione.studente.UID <> lezioneTarget.tutor.UID **Postcondizioni:**

	context GestioneLezioneDAO:: prenotaLezione(PrenotazioneDTO prenotazione) → boolean post: result = true implies (self.prenotazioni → includes(prenotazione) AND self.lezioni → select(l | l.idLezione = prenotazione.lezione.idLezione).stato = StatoLezione::PRENOTATA)
+annullaPrenotazione(int idPrenotazione) → boolean	**Descrizione:** Annulla una prenotazione esistente e libera lo slot della lezione associata (riportandola a PIANIFICATA). **Precondizioni:** context GestioneLezioneDAO:: annullaPrenotazione(int idPrenotazione) → boolean pre: let prenotazione = self.getPrenotazioneById(idPrenotazione) in prenotazione <> null AND prenotazione.stato = StatoPrenotazione::ATTIVA **Postcondizioni:** context GestioneLezioneDAO:: annullaPrenotazione(int idPrenotazione) → boolean post: let p = self.getPrenotazioneById(idPrenotazione) in p.stato = StatoPrenotazione::ANNULLATA AND p.lezione.stato = StatoLezione::PIANIFICATA
+getPrenotazioniByStudente(int idStudente) → List<PrenotazioneDTO>	**Descrizione:** Recupera lo storico di tutte le prenotazioni effettuate da uno studente (attive, concluse, annullate). **Precondizioni:** context GestioneLezioneDAO:: getPrenotazioniByStudente(int idStudente) → List<PrenotazioneDTO> pre: idStudente > 0 **Postcondizioni:** context GestioneLezioneDAO:: getPrenotazioniByStudente(int idStudente) → List<PrenotazioneDTO> post: result → forAll(p | p.studente.UID = idStudente)

+getLezioneById(int id) → LezioneDTO	**Descrizione**:
	Recupera i dettagli di una singola lezione dato il suo ID.
	
	Postcondizioni:
	context GestioneLezioneDAO:: getLezioneById(int id) → LezioneDTO
	post: if self.lezioni → exists(l | l.idLezione = id)
	then result = self.lezioni → select(l | l.idLezione = id) → first()
	else
	result = null
	endif

- **Classe GestioneLezioneController (Control)**
  - Descrizione**:** Componente logico che raggruppa le Servlet del package it.unisa.educat.controller.gestionelezione. Gestisce l'intero ciclo di vita di una lezione: dalla pubblicazione dell'annuncio da parte del Tutor, alla ricerca e prenotazione da parte dello Studente, fino alla gestione dello storico e degli annullamenti. Gestisce l'interazione tra le JSP (View) e il GestioneLezioneDAO (Model), garantendo la validazione dei vincoli temporali e di ruolo.
  - Operazioni**:**

Operazione	Servlet Implementativa	Contratto (OCL)
+cercaLezione(criteriRicerca criteri) → List<LezioneDTO>	CercaLezioneServlet (doGet)	**Descrizione:** Recupera le lezioni disponibili filtrandole per materia, città, prezzo e modalità. **Precondizioni:** context GestioneLezioneController:: cercaLezione(criteriRicerca criteri) → List<LezioneDTO> pre: criteri <> null **Postcondizioni:** context GestioneLezioneController:: cercaLezione(criteriRicerca criteri) → List<LezioneDTO> post: result → forAll(l | l.stato = StatoLezione::PIANIFICATA AND l.dataInizio > DateTime::now() AND (criteri.materia <> null implies l.materia = criteri.materia) AND (criteri.prezzoMax <> null implies l.prezzo ≤ criteri.prezzoMax))
+pubblicaAnnuncio(LezioneDTO lezione, UtenteDTO Tutor)	PubblicaAnnuncioServlet (doPost)	**Descrizione**: Permette a un Tutor di creare una nuova lezione (stato PIANIFICATA). **Precondizioni:** context GestioneLezioneController:: pubblicaAnnuncio(LezioneDTO lezione, UtenteDTO Tutor) pre: tutor <> null AND tutor.tipo = TipoUtente::TUTOR AND lezione.prezzo > 0 AND lezione.dataInizio < lezione.dataFine AND !GestioneLezioneDAO.hasTutorLezioneInFasciaOraria(tutor.UID, lezione.dataInizio, lezione.dataFine) **Postcondizioni:** context GestioneLezioneController::

		pubblicaAnnuncio(LezioneDTO lezione, UtenteDTO Tutor) post: GestioneLezioneDAO.lezioni → includes(lezione) AND lezione.stato = StatoLezione::PIANIFICATA
+prenotaLezione(int idLezione, UtenteDTO studente)	PrenotaLezioneServlet (doPost)	**Descrizione**: Gestisce la prenotazione: verifica disponibilità, processa il pagamento (simulato) e cambia lo stato della lezione. **Precondizioni:** context GestioneLezioneController:: prenotaLezione(int idLezione, UtenteDTO studente) pre: studente <> null AND (studente.tipo = TipoUtente::STUDENTE or studente.tipo = TipoUtente::GENITORE) AND let l = GestioneLezioneDAO.getLezioneById(idLezione) in l <> null AND l.stato = StatoLezione::PIANIFICATA AND l.tutor.UID <> studente.UID **Postcondizioni:** context GestioneLezioneController:: prenotaLezione(int idLezione, UtenteDTO studente) post: GestioneLezioneDAO.prenotazioni → exists(p | p.lezione.idLezione = idLezione AND p.studente.UID = studente.UID) AND GestioneLezioneDAO.lezioni → select(l | l.idLezione = idLezione).stato = StatoLezione::PRENOTATA
+annullaPrenotazione(int idPrenotazione, UtenteDTO richiedente)	AnnullaPrenotazioneServlet (doPost)	**Descrizione**: (Studente) Annulla una prenotazione attiva, liberando lo slot della lezione. **Precondizioni:** context GestioneLezioneController:: annullaPrenotazione(int idPrenotazione, UtenteDTO richiedente) pre: let p = GestioneLezioneDAO.getPrenotazioneById(idPrenotazione) in p <> null AND p.studente.UID = richiedente.UID AND p.stato = StatoPrenotazione::ATTIVA AND p.lezione.dataInizio > DateTime::now().plusHours(24)

		Postcondizioni: context GestioneLezioneController:: annullaPrenotazione(int idPrenotazione, UtenteDTO richiedente) post: let p = GestioneLezioneDAO.getPrenotazioneById(idPrenotazione) in p.stato = StatoPrenotazione::ANNULLATA AND p.lezione.stato = StatoLezione::PIANIFICATA

+getStoricoLezioni(UtenteDTO utente) → List<LezioneDTO>	StoricoLezioniServlet (doGet)	**Descrizione:**
		Restituisce la lista delle lezioni (prenotate o erogate) dell'utente.
		Nota: Aggiorna automaticamente lo stato delle lezioni passate a CONCLUSA.
		
		Precondizioni:
		context GestioneLezioneController::
		getStoricoLezioni(UtenteDTO utente) → List<LezioneDTO>
		pre: utente <> null
		
		Postcondizioni:
		context GestioneLezioneController::
		getStoricoLezioni(UtenteDTO utente) → List<LezioneDTO>
		post: if (utente.tipo = TipoUtente::TUTOR) then
		result → forAll(l : LezioneDTO | l.tutor.UID = utente.UID)
		else
		result → forAll(p : PrenotazioneDTO | p.studente.UID = utente.UID)
		endif

+getInfoLezione(int idLezione) → LezioneDTO	InfoLezioneServlet (doGet)	**Descrizione**: Recupera i dettagli di una singola lezione per la pagina di checkout/dettaglio. **Precondizioni:** context GestioneLezioneController:: getInfoLezione(int idLezione) → LezioneDTO pre: idLezione > 0 **Postcondizioni:** context GestioneLezioneController:: getInfoLezione(int idLezione) → LezioneDTO post: let lezione = GestioneLezioneDAO.getLezioneById(idLezione) in result = lezione AND result <> null implies session.getAttribute("lezioneCheckout") = result

+cancellaLezione(int idLezione, UtenteDTO tutorRichiedente)	CancellaLezioneServlet (doPost)	**Descrizione**: (Tutor) Rimuove una lezione pianificata che non è stata ancora prenotata. **Precondizioni:** context GestioneLezioneController:: cancellaLezione(int idLezione, UtenteDTO tutorRichiedente) pre: let l = GestioneLezioneDAO.getLezioneById(idLezione) in l <> null AND l.tutor.UID = tutorRichiedente.UID AND l.stato = StatoLezione::PIANIFICATA **Postcondizioni:** context GestioneLezioneController:: cancellaLezione(int idLezione, UtenteDTO tutorRichiedente) post: !GestioneLezioneDAO.lezioni → exists(l | l.idLezione = idLezione)

3.3 Gestione Segnalazione

Questo sottosistema permette la moderazione degli utenti tramite segnalazioni.

- **Classe Segnalazione (Model)**
  - Descrizione: Rappresenta una segnalazione inviata da un utente nei confronti di un altro (es. studente verso tutor o viceversa) per comportamenti scorretti.
  - Invarianti: context SegnalazioneDTO inv: self.descrizione <> null AND self.descrizione <> "" AND self.segnalante <> null AND self.segnalato <> null AND self.segnalante.UID <> self.segnalato.UID
  - Attributi:
    - -idSegnalazione: int
    - -descrizione: String
    - -stato: StatoSegnalazione, enum(“ATTIVA”, “RISOLTA”)
    - -segnalante: UtenteDTO
    - -segnalato: UtenteDTO
  - Operazioni: Getters e Setters per ciascun attributo.

- **Classe SegnalazioneDAO (Persistence)**
  - Descrizione: Classe DAO per la gestione delle segnalazioni nel database. Fornisce metodi per salvare nuove segnalazioni, recuperare l'intera lista (con join sui dati degli utenti coinvolti) e gestire la risoluzione o eliminazione delle stesse.
  - Operazioni:

Operazione	Contratto (OCL)
+doSave(SegnalazioneDTO segnalazione) →	**Descrizione:**

boolean	Salva una nuova segnalazione nel database. **Precondizioni:** context GestioneSegnalazioneDAO:: doSave(SegnalazioneDTO segnalazione) → boolean pre: segnalazione <> null AND segnalazione.descrizione <> null AND segnalazione.idSegnalante > 0 AND segnalazione.idSegnalato > 0 **Postcondizioni:** context GestioneSegnalazioneDAO:: doSave(SegnalazioneDTO segnalazione) → boolean post: result = true implies (self.segnalazioni → includes(segnalazione) AND segnalazione.idSegnalazione > 0)
+doRetrieveAll() → List<SegnalazioneDTO>	**Descrizione:** Restituisce tutte le segnalazioni presenti nel sistema, includendo i dettagli (nome/cognome) di segnalante e segnalato. **Postcondizioni:** context GestioneSegnalazioneDAO:: doRetrieveAll() → List<SegnalazioneDTO> post: result <> null AND result → forAll(s | self.segnalazioni → includes(s)) AND result → size() = self.segnalazioni → size()
+setAsSolved(idSegnalazione: int) → boolean	**Descrizione:** Imposta lo stato della segnalazione specificata a "RISOLTA" (chiusura del ticket). **Precondizioni:** context GestioneSegnalazioneDAO:: setAsSolved(idSegnalazione: int) → boolean pre: self.segnalazioni → exists(s | s.idSegnalazione = idSegnalazione) **Postcondizioni:** context GestioneSegnalazioneDAO:: setAsSolved(idSegnalazione: int) → boolean post: let s = self.segnalazioni → select(s | s.idSegnalazione = idSegnalazione) → first() in result = true implies s.stato = StatoSegnalazione::RISOLTA
+doDelete(idSegnalazione: int) → boolean	**Descrizione:** Elimina definitivamente una segnalazione dal

	database. **Precondizioni:** context GestioneSegnalazioneDAO:: doDelete(idSegnalazione: int) → boolean pre: self.segnalazioni → exists(s | s.idSegnalazione = idSegnalazione) **Postcondizioni:** context GestioneSegnalazioneDAO:: doDelete(idSegnalazione: int) → boolean post: result = true implies !self.segnalazioni → exists(s | s.idSegnalazione = idSegnalazione)

- **Classe SegnalazioneController (Control)**
  - Descrizione: Componente logico che raggruppa le Servlet del package it.unisa.educat.controller.gestionesegnalazione. Gestisce il flusso delle segnalazioni tra utenti e amministratori. Include l'invio asincrono (JSON) delle segnalazioni e le operazioni di moderazione riservate agli amministratori.

Operazione	Servlet Implementativa	Contratto (OCL)
+inviaSegnalazione(int idSegnalante, int idSegnalato, String motivo) → JSONResponse	InviaSegnalazioneServlet (doPost)	**Descrizione**: Riceve i dati in POST, valida che un utente non si segnali da solo e restituisce una risposta JSON (Success/Error). Non effettua redirect. **Precondizioni:** context GestioneSegnalazioneController:: inviaSegnalazione(int idSegnalante, int idSegnalato, String motivo) → JSONResponse pre: segnalante <> null AND idSegnalato <> null AND descrizione <> null AND descrizione <> "" AND segnalante.UID <> idSegnalato **Postcondizioni:** context GestioneSegnalazioneController:: inviaSegnalazione(int idSegnalante, int idSegnalato, String motivo) → JSONResponse post: let nuovaSegnalazione = GestioneSegnalazioneDAO.segnalaz

		ioni → last() in nuovaSegnalazione.segnalante.UID = segnalante.UID AND nuovaSegnalazione.segnalato.UID = idSegnalato AND nuovaSegnalazione.stato = StatoSegnalazione::ATTIVA AND result.json.success = true
+getListaSegnalazioni(UtenteDTO admin) → List<Segnalazione>	ListaSegnalazioniServlet (doGet)	**Descrizione**: (Admin) Recupera tutte le segnalazioni dal DB, filtra solo quelle in stato ATTIVA e le passa alla dashboard di amministrazione (homeAdmin.jsp). **Precondizioni:** context GestioneSegnalazioneController::getListaSegnalazioni(UtenteDTO admin) → List<Segnalazione> pre: admin <> null AND admin.tipo = TipoUtente::AMMINISTRATORE **Postcondizioni:** context GestioneSegnalazioneController::getListaSegnalazioni(UtenteDTO admin) → List<Segnalazione> post: result → forAll(s | s.stato = StatoSegnalazione::ATTIVA)
+risolviSegnalazioni(int idSegnalazione, UtenteDTO admin)	RisolviSegnalazioneServlet (doPost)	**Descrizione:** (Admin) Gestisce la risoluzione di una segnalazione chiamANDo il metodo setAsSolved del DAO. Reindirizza alla lista aggiornata dopo l'operazione._ **Precondizioni:** context GestioneSegnalazioneController::risolviSegnalazioni(int idSegnalazione, UtenteDTO admin) pre: admin <> null AND admin.tipo = TipoUtente::AMMINISTRATORE AND

		GestioneSegnalazioneDAO.segnalazioni → exists(s | s.idSegnalazione = idSegnalazione) **Postcondizioni:** context GestioneSegnalazioneController::risolviSegnalazioni(int idSegnalazione, UtenteDTO admin) post: let s = GestioneSegnalazioneDAO.segnalazioni → select(s | s.idSegnalazione = idSegnalazione) → first() in s.stato = StatoSegnalazione::RISOLTA

4. Glossario

Termine	*Significato*
Apache Tomcat	Web server e servlet container open source. Implementa le specifiche Java Servlet e JavaServer Pages.
Boundary Condition	Condizione limite o caso eccezionale nel funzionamento del sistema, come l'avvio (start-up), lo spegnimento (shut-down) o la gestione degli errori (failures).
Controller	Nel pattern MVC, è il componente che riceve l'input dell'utente (tramite la View), lo elabora (spesso interagendo con il Model) e determina la risposta o la vista successiva. In EduCat è implementato tramite Java Servlet.
Event-Driven	Paradigma di programmazione in cui il flusso del programma è determinato da eventi come le azioni dell'utente (click del mouse, invio di moduli) o messaggi da altri programmi.
Java Servlet	Componente software lato server che estende le capacità di un server web per gestire il modello richiesta-risposta.
Model	Nel pattern MVC, rappresenta la struttura dei dati e la logica di business dell'applicazione, indipendente dall'interfaccia utente. Gestisce l'interazione diretta con il database.
Thin Client	Architettura client-server in cui il client (in questo caso il browser) ha funzionalità ridotte e dipende principalmente dal server centrale per l'elaborazione dei dati e la logica applicativa.
Three-Tier	Architettura software multi-tier in cui i processi logici sono divisi in tre

Architecture	livelli fisici separati: Presentazione (Client), Logica Applicativa (Web Server) e Dati (Database Server).
View	Nel pattern MVC, è il componente responsabile della visualizzazione dei dati all'utente. In EduCat è implementato tramite pagine JSP.