Università degli Studi di Salerno
Corso di Ingegneria del Software

EduCat
Object Design Document
Versione 1.2

Data: 15/01/2026

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	Tutto il team
14/01/2026	1.1	Modifiche all'Overview dei pacchetti e aggiunto Glossario	Anna Chiara Esposito
15/01/2026	1.2	Modifiche alle Inerfacce delle Classi, modifiche all'indice	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.
- **Design Pattern**: soluzioni tipiche ai problemi più comuni nella progettazione del software. Alcuni esempi sono il Facade Pattern per Pattern Strutturali, Command Pattern per Pattern Comportamentali e Abstract Factory per Pattern Creazionali.
- **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 v1.4
- SDD_EduCat v1.2
- 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 MySQL tramite JDBC.	it.unisa.educat.model
it.unisa.educat.control	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 i bean forniti dal control.

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 classi Entità/Bean come Utente.java, Lezione.java, Recensione.java)
- src/main/java/it/unisa/educat/dao/ (Contiene le classi per l'accesso ai dati come GestioneUtenzaDAO.java, GestioneLezioneDAO.java)
- src/main/java/it/unisa/educat/control/ (Contiene le Servlet come GestioneUtenzaController.java, GestioneLezioneController.java)
- src/main/webapp/ (Contiene le pagine JSP, i fogli di stile CSS e le risorse statiche)

3. Interfacce delle classi

Questa sezione descrive le interfacce delle classi principali raggruppate per sottosistema. Per ogni sottosistema vengono dettagliate le classi Entità (Model), le interfacce 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.

- **Class Utente (Model)**
  - Rappresenta l'utente generico nel sistema.
  - Invarianti: context Utente inv: self.email < > null and self.password < > null
  - Attributi:
    - nome: String
    - cognome: String
    - email: String
    - password: String (Hashed)

        - dataNascita: Date
        - indirizzo: String (Composto da Via, Civico, Città, CAP)

- **Interfaccia GestioneUtenzaDAO (Persistence)**
    - Interfaccia per le operazioni CRUD sul database.
    - Operazioni:
        - DoSave(Utente u) throws SQLException
        - DoRetriveByEmail(String email) → Utente
        - doDelete(int id) throws SQLException

- **Classe GestioneUtenzaController (Control)**
    - Gestione la logica applicativa per gli utenti.

Servizio/Operazione	Contratto (OCL)
+login(String email, String password) → Utente	**Abstract**: Verifica le credenziali e restituisce l'utente corrispondente o null/eccezioni in caso di fallimento. **Postcondizioni**: context GestioneUtente::login(String email, String password) post: self.utenti → exists(u| u.email = email and u.password = password and result = u) or result = null
+registraUtente(Utente nuovoUtente)	**Abstract**: Riceve i dati di un nuovo utente e lo memorizza nel sistema **Precondizioni:** context GestioneUtente::registraUtente(Utente nuovoUtene) pre: !self.utenti → includes(nuovoUtente) **Postcondizioni:** context GestioneUtente:: registraUtente(Utente nuovoUtente) post: self.utenti → includes(nuovoUtente) and nuovoUtente.stato = CONFIRMED
+logout(session)	
+eliminaAccount(idUtente)	
+modificaProfilo(Integer idUtente, Utente datiNuovi) → Utente	**Abstract:** Aggiorna i dati dell'utente. **Precondizioni:** context GestioneUtente::modificaProfilo(Integer idUtente, Utente datiNuovi) → Utente pre: self.utenti → exists(u| u.id = idUtente) **Postcondizioni:** context GestioneUtente::modificaProfilo(Integer idUtente, Utente datiNuovi) → Utente post: self.utenti → includes(datiNuovi)

+cambiaRuolo(idUtente, nuovoRuolo)	
+getListaUtenti(filtri) → List<Utente>	**Abstract:** Recupera un preciso insieme di utenti in base al filtro selezionato **Postcondizioni:** context GestioneUtenza:: getListaUtenti(filtri) → List(Utente) post: self.utenti → exists(u|

3.2 Gestione Lezione

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

- **Class Lezione (Model)**
  - Rappresenta una lezione offerta o prenotata.
  - **Invarianti**: Context Lezione inv: self.prezzo >= 0
  - Attributi Pubblici:
    - idLezione: int
    - materia: String
    - data: Date
    - durata: float
    - prezzo: float
    - modalitaLezione: String (Enum: ONLINE/PRESENZA)
    - tutor: Tutor
    - città: String

- **Class Prenotazione (Model)**
  - Associa uno studente a una lezione prenotata.
  - Invarianti: context Prenotazione inv: self.stato = 'ATTIVA' or 'ANNULLATA' or 'CONCLUSA'
  - Attributi Pubblici:
    - idPrenotazione: int
    - dataPrenotazione: Date
    - stato: Enum (ATTIVA, ANNULLATA, CONCLUSA)
    - idStudente: int
    - idLezione: int

- **Interfaccia GestioneLezioneDAO (Persistence)**
  - Operazioni:
    - doSaveLezione(Lezione l)
    - doRetrieveByCriteria(CriteriRicerca c) → List<Lezione>
    - doSavePrenotazione(Prenotazione p)
    - doUpdateStatoPrenotazione(int id, String stato)

- **Classe GestioneLezioneController (Control)**
  - Gestisce la logica applicativa per le lezioni.

Servizio/Operazione	Contratto (OCL)
+cercaLezione(criteriRicerca)	**Abstract:**

+pubblicaAnnuncio(Lezione lezione)	Abstract:
+prenotaLezione(int idLezione, int idStudente, Date slot)	Abstract:
+annullaPrenotazione(int idPrenotazione)	Abstract:
+getStoricoLezioni(int idUtente) → List<Lezione>	Abstract:

3.3 Gestione Segnalazione

Questo sottosistema permette la moderazione degli utenti tramite segnalazioni.

- **Classe Segnalazione (Model)**
  - Attributi:
    - idSegnalazione: int
    - descrizione: String
    - idSegnalante: int
    - idSegnalato: int

- **Interface SegnalazioneDAO (Persistence)**
  - Operazioni:
    - doSave(Segnalazione s)
    - doRetrieveAll() → List<Segnalazione>

- **Classe SegnalazioneController (Control)**
  - Gestisce la logica applicativa per le lezioni.

Servizio/Operazione	Contratto (OCL)
+inviaSegnalazione(int idSegnalante, int idSegnalato)	Abstract:
+getListaSegnalazioni() → List<Segnalazione>	Abstract:
+risolviSegnalazioni(int idSegnalazione)	Abstract:

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 Architecture	Architettura software multi-tier in cui i processi logici sono divisi in tre 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.