UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze

Dipartimento di Ingegneria dell'Informazione

LEARN - IT

Autore:
Lorenzo Ciabatti

N° Matricola:
6292744

Corso principale:
Ingegneria del Software

Docente corso:
Enrico Vicario

Contents

1 Introduzione

1.1 Statement

Il presente programma è progettato per gestire un centro di formazione su tematiche legate al settore IT, offrendo funzionalità avanzate per l'organizzazione e la gestione delle iscrizioni, nonché per la visualizzazione dei dati anagrafici degli impiegati partecipanti. Le aziende possono consultare il calendario dei corsi di formazione previsti e accedere ai contenuti multimediali pubblicati dai formatori.

- **Amministratore**: è responsabile della creazione, modifica ed eventuale eliminazione dei corsi di formazione, assegnando a ciascuno una data, un orario e una breve descrizione. Inoltre, gestisce la pianificazione dei turni dei formatori e invia promemoria alle aziende in merito al pagamento delle quote di iscrizione. Al termine della creazione o modifica di un corso, il sistema invia automaticamente una notifica sia alle aziende sia ai formatori, garantendo una comunicazione puntuale. In modo analogo, ogni volta che i turni vengono assegnati o modificati, i formatori coinvolti ricevono una notifica.
- **Formatori**: hanno accesso alla visualizzazione dei propri turni di lavoro e all'elenco completo degli impiegati iscritti. Per ciascun impiegato è possibile consultare le informazioni anagrafiche e altri dati utili, come ruolo, età e azienda di appartenenza. I formatori possono inoltre caricare, e se necessario rimuovere, materiali didattici (slide e video) relativi ai corsi.
- **Aziende**: si occupano dell'iscrizione dei propri impiegati ai corsi di formazione, fornendo i dati richiesti. Possono consultare in qualsiasi momento il calendario dei corsi pianificati e il materiale multimediale caricato dai formatori. Il pagamento della quota di iscrizione può essere effettuato anche successivamente. È prevista una tariffa intera per ciascun impiegato, con una riduzione applicata in caso di iscrizione multipla da parte della stessa azienda.

1.2 Tecnologie e Strumenti Utilizzati

Per lo sviluppo del presente progetto sono stati adottati strumenti e tecnologie, selezionati per garantire efficienza, scalabilità e qualità del software.

- **Ambiente di sviluppo:** Intellij IDEA, integrato con GitHub Copilot, è stato utilizzato come IDE principale per scrivere, completare e ottimizzare il codice Java, migliorando la produttività grazie all'automazione intelligente.
- **Linguaggio di programmazione:** Java è stato scelto per la sua robustezza, portabilità e ampia diffusione nel contesto enterprise.
- **Accesso al database:** La connessione con il database è stata gestita tramite JDBC (Java Database Connectivity), permettendo un'interazione diretta e performante con il database relazionale.
- **Database:** PostgreSQL 17 è stato impiegato come sistema di gestione del database (DBMS), garantendo affidabilità, integrità e supporto avanzato per i dati strutturati.
- **Testing:** I test unitari sono stati realizzati con JUnit, assicurando il corretto funzionamento delle singole componenti software e facilitando la manutenzione del codice.
- **Modellazione:** Per la progettazione del sistema sono stati utilizzati diagrammi UML, Use Case, ER realizzati con l'ausilio di StarUML, strumento professionale per la modellazione visuale.
- **Mockup:** Balsamiq è stato adottato per la realizzazione dei mockup delle interfacce utente, permettendo una rapida definizione e validazione del design.
- **Integrazione AI:** È stata implementata un'integrazione con ChatGPT per generare dati di test automatizzati, riducendo la necessità di popolamento manuale del database e migliorando l'efficienza dei test.
- **Controllo versione:** Il codice sorgente è gestito tramite un repository GitHub, accessibile all'indirizzo: github.com/ciabatti/LEARN-IT.

- **Redazione documentale:** La relazione tecnica è stata stesa utilizzando Overleaf, piattaforma collaborativa basata su LaTeX, per garantire un'elevata qualità tipografica e facilità di revisione.

2 Progettazione

2.1 Use Case Diagram

L'applicativo prevede tre attori principali: l'**Amministratore**, i **Formatori** e le **Aziende**.

L'**Amministratore** è responsabile della gestione complessiva del sistema, incluse le attività di *pianificazione*, *modifica* e *eliminazione* dei corsi di formazione, nonché della *gestione* dei turni dei formatori. Supervisiona inoltre la comunicazione con le aziende, in particolare attraverso l'invio di *notifiche* relative a iscrizioni e pagamenti.

Le **Aziende**, in qualità di enti iscrittori, si occupano dell'*iscrizione* dei propri impiegati ai corsi, fornendo le informazioni richieste. Hanno accesso al *calendario delle attività formative* e possono visualizzare i *materiali multimediali* caricati dai formatori.

I **Formatori** possono consultare i propri turni di lavoro e l'elenco degli impiegati iscritti ai corsi di loro competenza. Per ogni partecipante è possibile visualizzare i dati anagrafici, il ruolo e l'azienda di provenienza. I formatori sono inoltre responsabili del *caricamento* e della *gestione* dei contenuti didattici multimediali (slide, video, ecc.) relativi ai corsi.

In figura 1 è mostrato il diagramma complessivo dei casi d'uso dell'applicativo. Le figure 2, 3 e 4 rappresentano i casi d'uso suddivisi per attore, fornendo una visione dettagliata e strutturata delle funzionalità associate a ciascun ruolo.

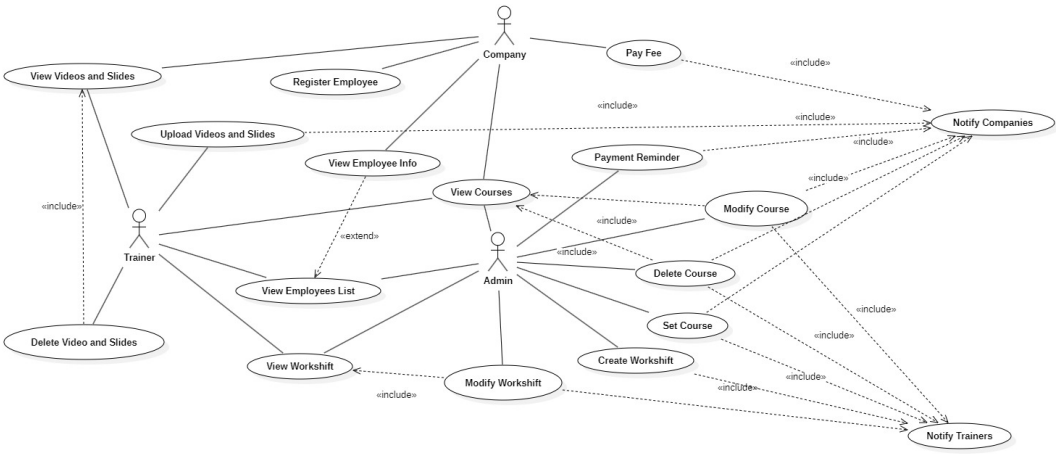

Figure 1: Use case diagram completo

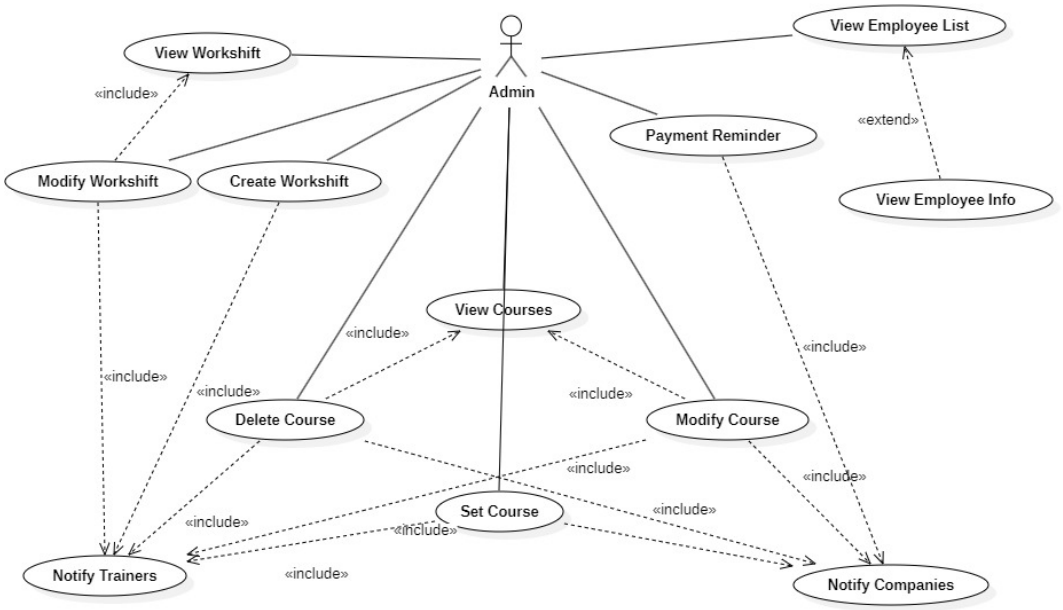

Figure 2: Use case diagram Admin

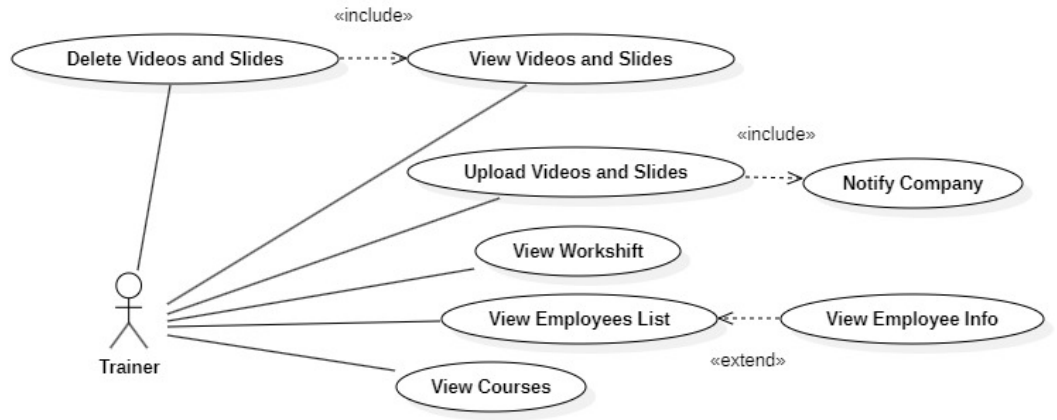

Figure 3: Use case diagram Trainer

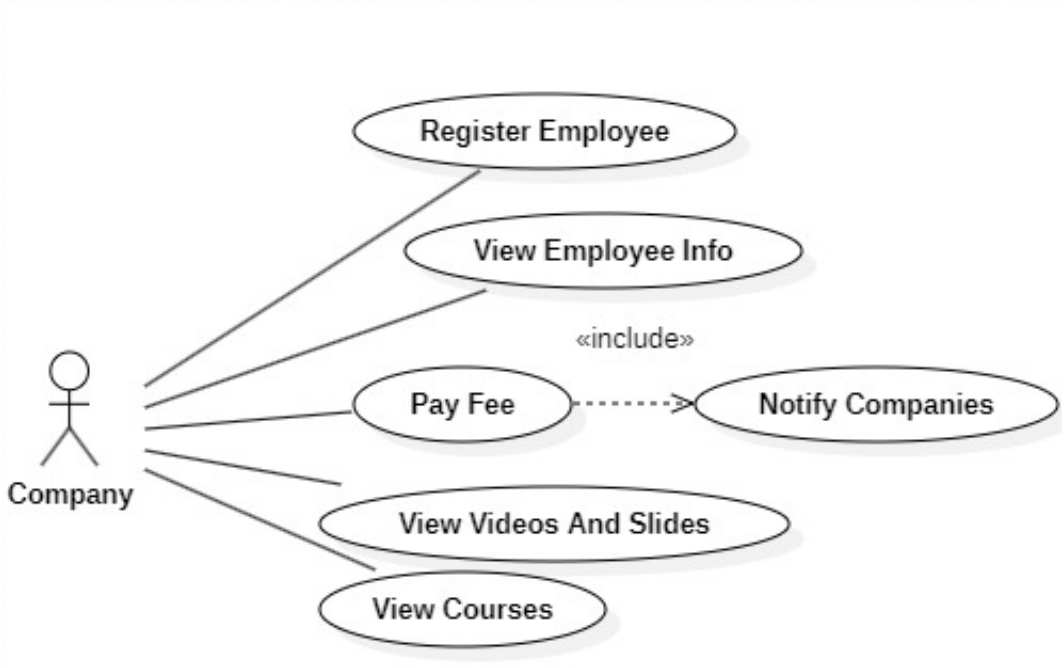

Figure 4: Use case diagram Company

2.2 Use case templates

Al fine di approfondire il comportamento del sistema in risposta alle interazioni degli utenti, sono stati selezionati e descritti, mediante l'utilizzo di template standard, alcuni casi d'uso rappresentativi dell'applicazione. In particolare, sono stati analizzati tre casi d'uso relativi all'**Amministratore**, due riguardanti i **Formatori** e due associati alle **Aziende**. Tali casi d'uso evidenziano le funzionalità principali offerte a ciascuna categoria di attori e permettono di delineare con maggiore precisione le responsabilità e i flussi operativi del sistema.

UC-1	Set Course
Level	User goal
Main Actors	Admin
Description	L'utente di tipo Admin sceglie un corso stabilendone giorno, orario e descrizione.
Basic Course	1. Utente accede alla pagina *CREATE COURSE* 2. Il sistema mostra un form da compilare con i campi: Date, Time, Description e Focus (il macro-argomento) 3. Una volta riempiti i campi, l'utente preme il pulsante *CREATE* 4. Se tutti i valori sono corretti, il sistema registra il corso e notifica alle Aziende e ai Formatori la presenza della nuova lezione
Alternative Course	4. Se i valori inseriti non hanno il formato richiesto o sono invalidi, il sistema lo segnala all'utente

Table 1: UC-1: Set Course (vedi figura: 5)

UC-2	Delete Course
Level	User goal
Main Actors	Admin
Description	L'utente di tipo Admin sceglie un corso stabilendone giorno, orario e descrizione.
Basic Course	1. Utente accede alla pagina *DELETE COURSE* 2. Il sistema mostra un una lista dei corsi programmati 3. L'utente seleziona il corso che vuole eliminare e conferma la scelta mediante il bottone *DELETE SELECTED COURSE* 4. Il sistema elimina il corso e notifica alle Aziende e ai Formatori l'eliminazione della lezione selezionata

Table 2: UC-2: Delete Course (vedi figura: 6)

UC-3	View Employee List
Level	User goal
Main Actors	Admin, Trainer
Description	L'utente di tipo Admin o Trainer visualizza la lista degli impiegati iscritti a un corso
Basic Course	1. Utente seleziona la voce *VIEW EMPLOYEES LIST* 2. Il sistema mostra una lista degli impiegati iscritti ai corsi di formazione

Table 3: UC-3: View Employee List (vedi figura: 7)

UC-4	Upload Videos and Slides
Level	User goal
Main Actors	Trainer
Description	L'utente di tipo Trainer carica le slide e i video relativi a un corso.
Basic Course	1. L'utente seleziona la voce *UPLOAD VIDEO AND SLIDES* dalla barra laterale 2. Il sistema mostra due file picker, uno per ogni tipologia di materiale 3. L'utente può scegliere dal proprio PC il materiale da caricare e lo carica mediante l'apposito bottone 4. Se il caricamento va a buon fine, le aziende vengono informate della presenza di nuovi contenuti.
Alternative Course	Se il caricamento non va a buon fine, l'utente viene notificato.

Table 4: UC-4: Upload Videos and Slides (vedi figura: 8)

UC-5	Pay Fee
Level	User goal
Main Actors	Company
Description	L'utente di tipo Company può vedere i pagamenti che deve ancora effettuare e successivamente saldare eventuali fatture.
Basic Course	1. L'utente seleziona *PAY FEE* nel menù laterale e visualizza la lista degli impiegati della propria azienda iscritti ai corsi, insieme allo stato delle fatture. 2. Selezionando un impiegato con saldo non ancora pagato, può procedere al pagamento tramite l'apposito bottone. 3. La tariffa per ogni incontro si riduce in funzione del numero di impiegati iscritti: è intera solo se l' azienda ha un unico partecipante. 4. Il sistema calcola automaticamente l'importo totale da versare.

Table 5: UC-5: Pay Fee (vedi figura: 9)

UC-6	Register Employee
Level	User goal
Main Actors	Company
Description	L'utente di tipo Company iscrive un impiegato a un corso.
Basic Course	1. L'utente seleziona la voce *REGISTER EMPLOYEE* dalla barra laterale. 2. Il sistema mostra un modulo in cui inserire le informazioni dell'impiegato e il numero di lezioni che seguirà. 3. Tramite il tasto *GO!* la registrazione viene effettuata.
Alternative Course	Se il modulo non è compilato correttamente, l'utente viene notificato.

Table 6: UC-6: Register Employee (vedi figura: 10)

2.3 Mockup dell'interfaccia utente

A completamento della progettazione dei casi d'uso, sono stati realizzati dei mockup per rappresentare graficamente l'interfaccia utente associata alle principali funzionalità del sistema. I prototipi hanno lo scopo di illustrare, in modo intuitivo e visuale, il comportamento atteso dell'applicativo in risposta alle interazioni degli attori.

Ogni schermata è stata sviluppata utilizzando lo strumento *Balsamiq Mockups*, con l'intento di mantenere una rappresentazione semplice ma efficace dei componenti grafici. I mockup sono stati progettati seguendo i casi d'uso precedentemente descritti, rispettando i flussi logici previsti e agevolando l'attività di sviluppo successiva.

Le figure che seguono mostrano i mockup relativi a ciascuno dei casi d'uso documentati nella sezione precedente.

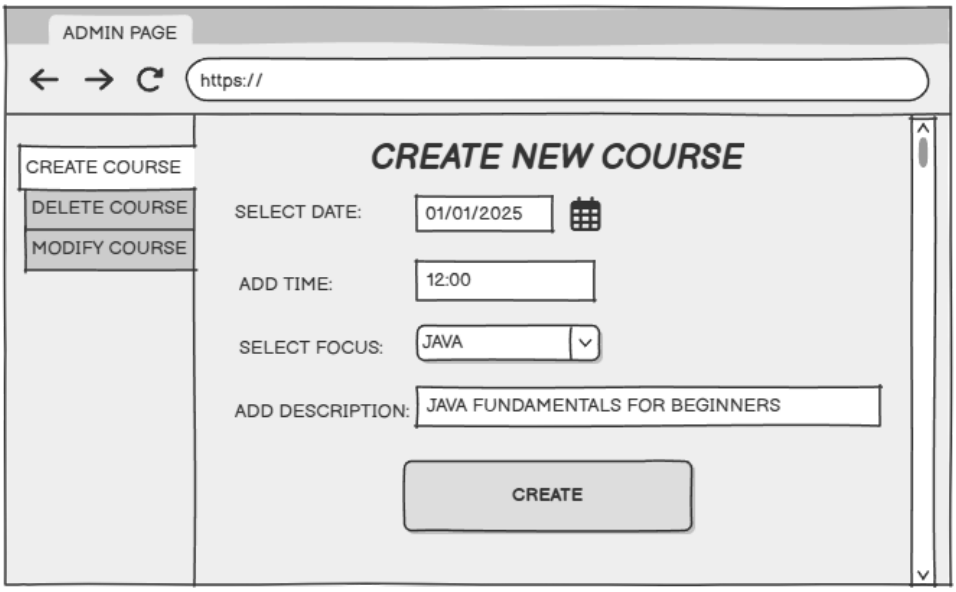

Figure 5: Mockup dell'interfaccia per la creazione di un corso (UC-1, vedi Tabella 1).

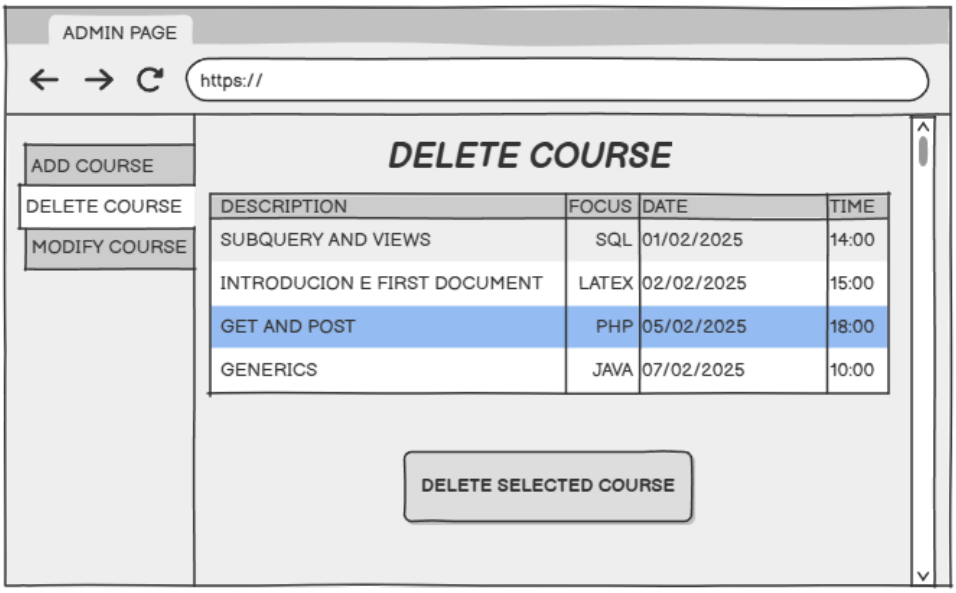

Figure 6: Mockup dell'interfaccia per l'eliminazione di un corso (UC-2, vedi Tabella 2).

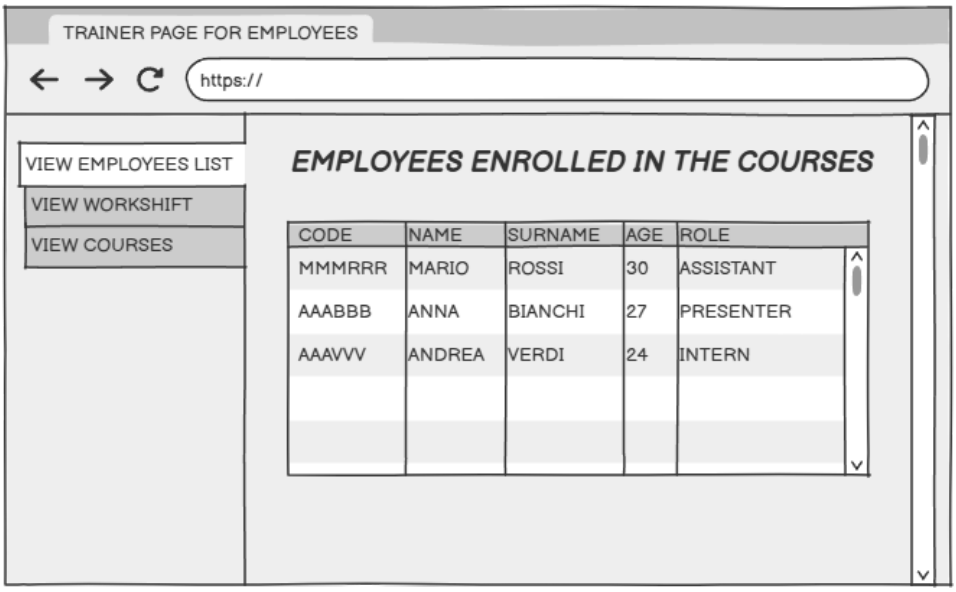

Figure 7: Mockup della visualizzazione degli impiegati iscritti (UC-3, vedi Tabella 3).

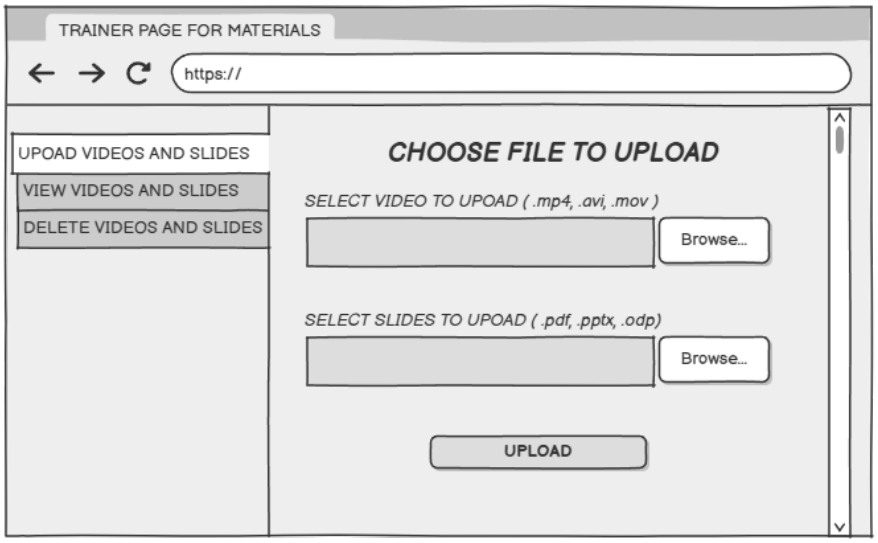

Figure 8: Mockup per il caricamento dei materiali da parte dei formatori (UC-4, vedi Tabella 4).

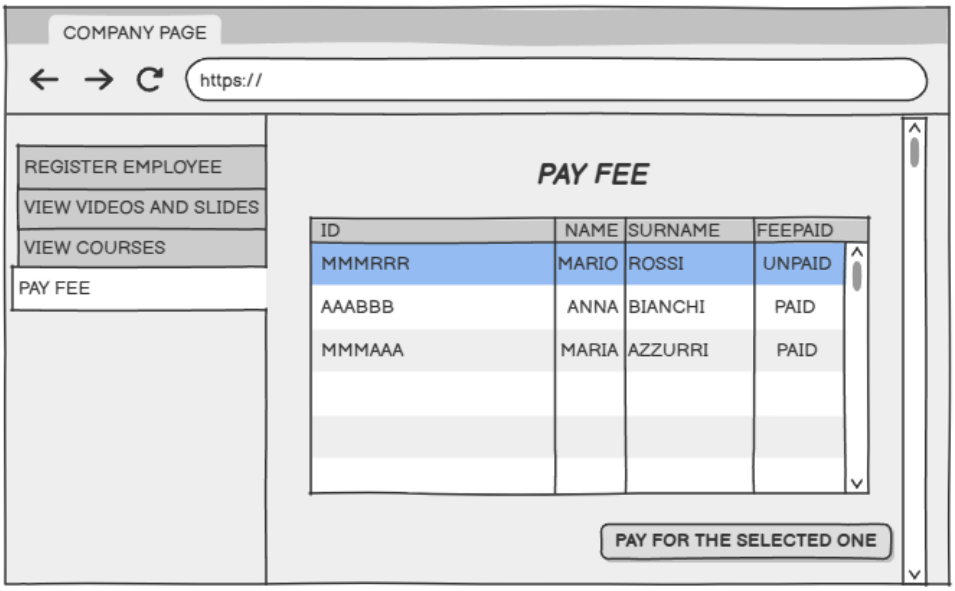

Figure 9: Mockup per la gestione e il pagamento delle quote di iscrizione (UC-5, vedi Tabella 5).

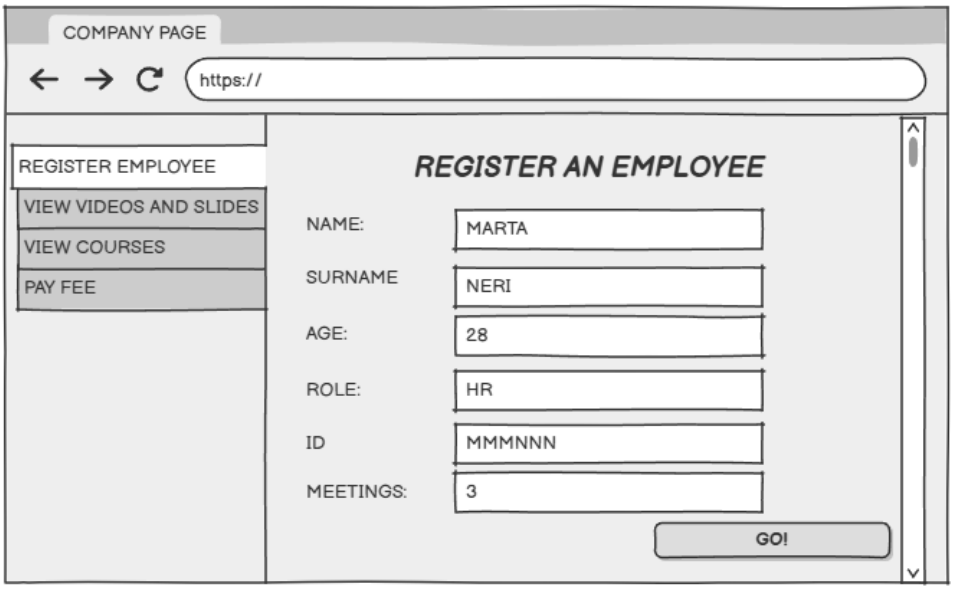

Figure 10: Mockup per l'iscrizione di un impiegato da parte di un'azienda (UC-6, vedi Tabella 6).

3 UML e Struttura

3.1 Package Diagram

Il diagramma dei package implementati è stato realizzato utilizzando *StarUML*, al fine di rappresentare graficamente la struttura statica del sistema e le relazioni tra i diversi componenti. L'architettura dell'applicativo è suddivisa in tre principali package, ognuno con responsabilità ben distinte secondo il principio della separazione delle responsabilità:

- **Business Logic**: contiene le classi deputate alla gestione della logica applicativa, inclusi i servizi e i casi d'uso.
- **Domain Model**: racchiude le entità principali del dominio, modellando i concetti chiave e le loro relazioni.
- **Object-Relational Mapping (ORM)**: comprende le componenti responsabili della persistenza, mappando le entità del dominio alle corrispondenti tabelle nel database relazionale.

Tale suddivisione favorisce una maggiore modularità e manutenibilità del sistema, rendendo più agevole l'estensione o la modifica di singole funzionalità.

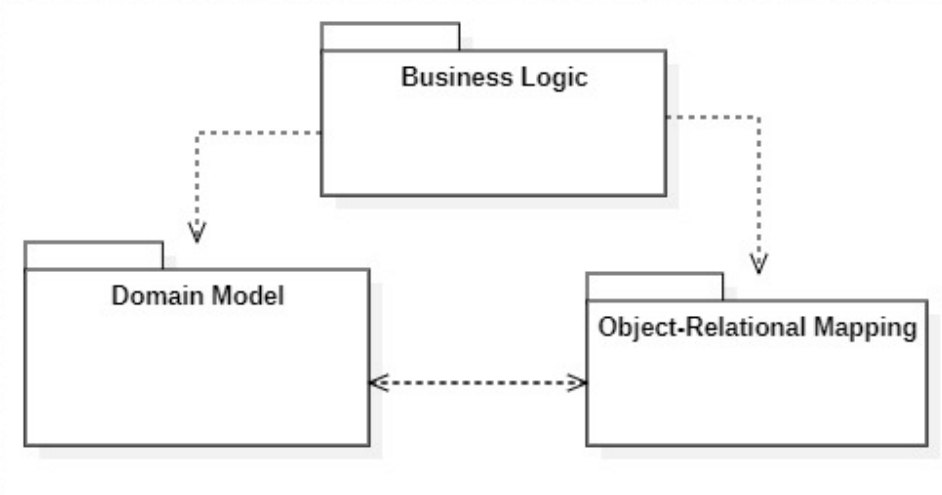

Figure 11: Package diagram di LEARN-IT

3.2 Class Diagram

In questa sezione vengono presentati i diagrammi delle classi relativi ai packages precedentemente mostrati nel *Package Diagram*. Ogni diagramma di classe illustra la struttura interna del rispettivo package, evidenziando le classi implementate, i loro attributi, metodi e le relazioni tra di esse (associazioni, ereditarietà, dipendenze).

L'obiettivo è fornire una visione dettagliata dell'architettura software a livello di progettazione, utile per comprendere l'organizzazione del codice, la responsabilità delle singole componenti e le interazioni tra i vari moduli del sistema.

3.2.1 Business Logic

Il package *Business Logic* raccoglie le classi responsabili dell'implementazione dei casi d'uso del sistema. Per ciascun attore è definito un apposito controller incaricato della gestione delle operazioni previste, coordinando l'interazione tra il *Domain Model* e i componenti di persistenza dell'*ORM*. All'interno del package è inoltre presente la classe `Notifier`, responsabile della gestione e dell'invio delle email di notifica agli utenti coinvolti.

Segue il diagramma delle classi contenute nella *Business Logic*:

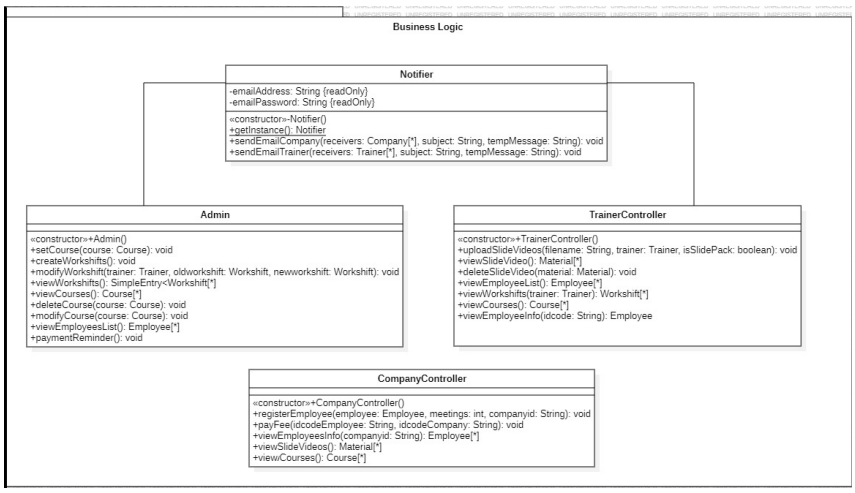

Figure 12: Class diagram del package Business Logic

3.2.1.1 Admin La classe `Admin` implementa le funzionalità riservate all'amministratore del sistema, consentendogli la gestione delle principali entità applicative. In particolare, il metodo *setCourse()*, visibile in figura 13, permette l'inserimento di un nuovo corso nel calendario, mentre *modifyCourse()* e *deleteCourse()* ne consentono rispettivamente la modifica e l'eliminazione. Ogni variazione sui corsi genera notifiche automatiche inviate tramite email alle aziende e ai formatori interessati.

Per quanto riguarda la pianificazione del personale, l'`Admin` può creare, modificare o eliminare i turni di lavoro dei formatori mediante i metodi *createWorkshifts()*, *modifyWorkshift()* e *deleteWorkshift()*. La visualizzazione dei turni programmati è invece gestita tramite *viewWorkshifts()*. Anche in questo caso, ogni aggiornamento genera una notifica automatica ai formatori coinvolti.

Completano le funzionalità della classe i metodi *viewEmployeesList()* e *viewEmployeeInfo()*, che consentono rispettivamente la visualizzazione dell'elenco dei dipendenti iscritti e dei relativi dati anagrafici e lavorativi. Infine, attraverso *paymentReminder()*, l'*Admin* può inviare promemoria di pagamento alle aziende.

```
public class Admin {
    private final Notifier notifier;

    public Admin() {
        notifier = Notifier.getInstance();
    }

    public void setCourse(Course course) throws SQLException, ParseException, MessagingException, ClassNotFoundException {
        CourseDAO courseDAO = new CourseDAO();
        courseDAO.insertCourse(course);

        CompanyDAO companyDAO = new CompanyDAO();

        notifier.sendEmailCompany(companyDAO.getAllCompanies() , "New course", "a new course has been created. You can check it on the website.");

        TrainerDAO trainerDAO = new TrainerDAO();

        notifier.sendEmailTrainer(trainerDAO.getAllTrainers(), "New course", "a new course has been created. You can check it on the website.");
    }
```

Figure 13: setCourse()

3.2.1.2 TrainerController La classe `TrainerController` fornisce le funzionalità operative a supporto delle attività dei formatori. Il metodo *viewWorkshifts()* consente la visualizzazione del proprio piano turni, mentre *viewCourses()* permette di consultare i corsi assegnati. I metodi *viewEmployeesList()* e *viewEmployeesInfo()* offrono strumenti per accedere all'elenco dei partecipanti e ai relativi dettagli.

Relativamente alla gestione dei materiali didattici, la classe include *uploadSlideVideo()* per il caricamento dei contenuti, la cui implementazione è visibile in figura 14, *deleteSlideVideo()* per la loro rimozione e *viewSlideVideo()* per la consultazione. Ogni modifica apportata ai materiali comporta l'invio automatico di notifiche email alle aziende, assicurando una comunicazione tempestiva e trasparente.

```
public void uploadSlideVideos(String filename, Trainer trainer, boolean isSlidePack) throws SQLException, IOException, ParseException,
                                                                                            MessagingException, ClassNotFoundException
{
    MaterialDAO materialDAO = new MaterialDAO();
    materialDAO.uploadMaterial(filename, trainer.getEmail(), isSlidePack);

    CompanyDAO companyDAO = new CompanyDAO();
    ArrayList<Company> companies = new ArrayList<>();
    companies = companyDAO.getAllCompanies();
    notifier.sendEmailCompany(companies, "New material", "new material has been uploaded. You can check it on the website.");
}
```

Figure 14: uploadSlideVideos()

3.2.1.3 CompanyController La classe `CompanyController` racchiude le funzionalità destinate alle aziende degli impiegati iscritti ai corsi. Il metodo *registerEmployee()* consente l'iscrizione del proprio dipendente al corso, il codice che implementa tale funzione è visibile successivamente in figura 15, mentre *payFee()* permette il versamento della quota di iscrizione. Le aziende possono inoltre consultare l'elenco dei corsi pianificate tramite il metodo *viewCourses()*, accedere alle dispense multimediali pubblicate con *viewSlideVideo()* e visualizzare le informazioni relative ai propri dipendenti attraverso *viewEmployeeInfo()*.

```
public void registerEmployee(Employee employee, int meetings, String companyid) throws SQLException, ClassNotFoundException {
    EmployeeDAO employeeDAO = new EmployeeDAO();
    ArrayList<Employee> empoyees = employeeDAO.getEmployeesByCompany(companyid);
    FeeStrategy feeStrategy;
    if(empoyees.isEmpty())
        feeStrategy = new SingleEmployeeFee();
    else if (empoyees.size() >= 1) {
        feeStrategy = new MultipleEmployeesFee();
        SubscriptionDAO subscriptionDAO = new SubscriptionDAO();
        subscriptionDAO.editFeeStrategy(empoyees.get(0).getIdcode(), feeStrategy);
    }
    else
        feeStrategy = new MultipleEmployeesFee();
    Subscription subscription = new Subscription(meetings, employee, feeStrategy, false);
    employee.setSubscription(subscription);
    employeeDAO.insertEmployee(employee, subscription, companyid);
}
```

Figure 15: registerEmployee()

3.2.1.4 Notifier La classe *Notifier* ha il compito di gestire l'invio di email personalizzate alle aziende (*Company*) e ai formatori (*Trainer*) del sistema. Implementata secondo il pattern *Singleton*, essa garantisce che esista una sola istanza condivisa, accessibile tramite il metodo *getInstance()*. Le funzionalità principali sono offerte dai metodi *sendEmailCompany()* e *sendEmailTrainer()*, che inviano messaggi email HTML contenenti un testo personalizzato, il nome del destinatario e un'immagine. La comunicazione avviene attraverso il protocollo SMTP, con autenticazione dell'utente mittente tramite un account Gmail configurato nella classe stessa. Questo componente centralizza e uniforma le notifiche inviate per eventi rilevanti, come ad esempio il pagamento delle quote, migliorando la comunicazione tra il sistema e i suoi utenti.

3.2.2 Domain Model

Il *Domain Model* rappresenta il nucleo concettuale dell'applicazione e contiene le classi che modellano i concetti principali del dominio. Queste classi sono indipendenti dalla logica di business e dalla persistenza, e riflettono fedelmente le entità, le regole e le relazioni proprie del sistema.

Di seguito è mostrato il diagramma delle classi che compongono il *Domain Model*:

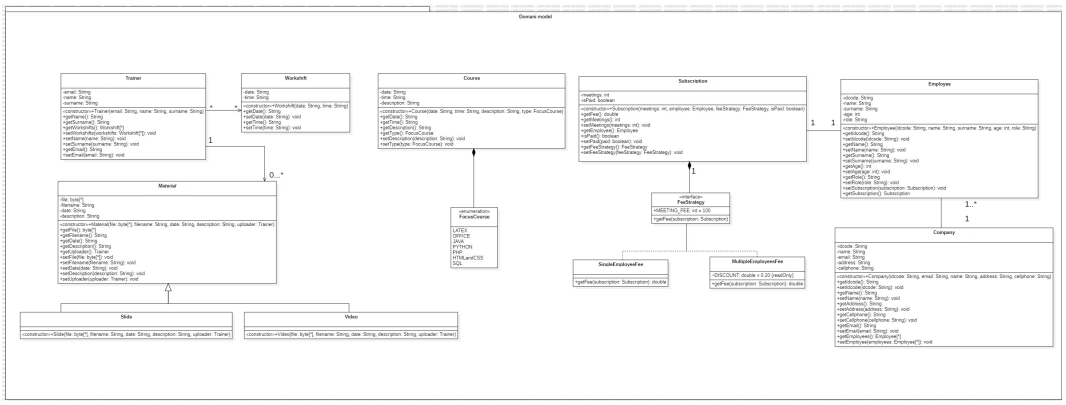

Figure 16: Class diagram del package Domain Model (è presente un'immagine di dimensione maggiore alla fine del documento, figura: 28)

3.2.2.1 Subscription Contiene le informazioni relative all'iscrizione di un impiegato ai corsi. Gli attributi sono il numero di incontri per cui si vuole effettuare l'iscrizione e un booleano che indica se la quota di essa è stata pagata. Sono presenti anche un riferimento all'oggetto di tipo *Employee* che rappresenta l'impiegato a cui è associata l'iscrizione e a un oggetto di tipo *FeeStrategy*: un'interfaccia spiegata nel paragrafo successivo, che indica se l'impiegato ha altri colleghi iscritti, in modo tale che sia possibile applicare la tariffa corretta per ciascuno dei due casi.

3.2.2.2 FeeStrategy, SingleEmployeeFee e MultipleEmployeeFee Queste tre classi realizzano il design pattern Strategy che permette di esporre diverse implementazioni di un metodo in base all'istanza che un oggetto realizza tra le diverse implementazioni dell'interfaccia principale; nel caso specifico, l'interfaccia principale è *FeeStrategy*, mentre le implementazioni sono *SingleEmployeeFee* e *MultipleEmployeeFee*. L'obiettivo è quello di dare diverse implementazioni del metodo *getFee()*, che restituisce la quota intera se il viene iscritto un solo impiegato, oppure la quota scontata se l'azienda decide di iscriverne più di uno. La scelta diquesta soluzione oltre sembrare adatta al caso, permette di dare informazioni aggiuntive per ogni implementazione, come per esempio l'attributo DISCOUNT nella classe *MultipleEmployeeFee*. Inoltre una soluzione di questo tipo permette una maggior apertura alla modifica e alla eventuale estensione con l'aggiunta di una nuova implementazione.

3.2.2.3 Employee È la classe che rappresenta ogni impiegato iscritto ai corsi. I suoi attributi esplicitano i riferimenti identificativi e personali dell' impiegato: *nome, cognome, codice identificativo, età e ruolo all'interno dell'azienda* . Sono inoltre presenti un riferimento a un oggetto di tipo *Subscription* che contiene dettagli specificatamente relativi all'iscrizione e un oggetto di tipo *Company* che rappresenta l'azienda per cui lavora.

3.2.2.4 Company Rappresenta un' azienda che intende iscrivere un impiegato a un corso di formazione. Ha attributi come un *codice identificativo, nome, indirizzo, numero di telefono, email* e una *lista* di oggetti di tipo *Employee* che rappresentano i dipendenti iscritti.

3.2.2.5 Trainer Classe utilizzata per incarnare un formatore che svolgerà un corso. Gli attributi principali di tale classe so *nome, cognome, email* e una *lista* di oggetti di tipo *Workshift* che definisce i suoi turni di lavoro.

3.2.2.6 Workshift La classe *Workshift* rappresenta un turno di lavoro. Essa è caratterizzata da attributi che definiscono la *data* e *orario* di inizio del corso, fondamentali per la pianificazione e la gestione delle attività dei formatori.

3.2.2.7 Material, Slide e Video È una classe astratta che modella un file multimediale caricato da un formatore, con l'obiettivo di renderlo fruibile. Gli attributi principali includono la *data* e l'*orario* di caricamento, il *nome del file*, un *array* di byte che rappresenta il contenuto effettivo del file, e un *riferimento* all'oggetto *Trainer* che identifica chi

ha inserito il contenuto. Le classi *Slide* e *Video* estendono *Material*, distinguendo così due diverse tipologie di contenuti multimediali, rispettivamente blocchi di slide e filmati.

3.2.2.8 Course La classe *Course* modella un corso svolto. Essa possiede attributi che indicano la *data* e l'*ora* in cui l'attività ha luogo, una *descrizione* testuale del corso stesso, e un riferimento a un oggetto di tipo *FocusCourse*, che categorizza il corso in una specifica tipologia.

3.2.2.9 FocusCourse *FocusCourse* è un tipo *enum* che elenca tutte le possibili categorie di corsi offerti. Questa implementazione è stata scelta per la sua semplicità, chiarezza e facilità di estensione futura, garantendo al contempo una tipizzazione sicura e un controllo rigoroso sulle categorie consentite.

3.2.3 Object-Relational Mapping (ORM)

Il package *Object-Relational Mapping* ha il compito di gestire la persistenza dei dati, fungendo da strato intermedio tra il *Domain Model* e il sistema di gestione del database relazionale (RDBMS). In particolare, contiene le classi DAO, ognuna delle quali è dedicata alla gestione della persistenza di una specifica entità del dominio.

Le classi DAO incapsulano la logica di accesso ai dati, fornendo metodi per operazioni CRUD (*Create*, *Read*, *Update*, *Delete*) e interfacciandosi direttamente con le tabelle del database. Questa separazione delle responsabilità consente di mantenere il *Domain Model* indipendente dai dettagli implementativi relativi alla persistenza, migliorando la manutenibilità e testabilità del sistema.

Tutte le interazioni con il database, come la registrazione di una nuova sottoscrizione, la modifica delle informazioni di un impiegato o il recupero dei corsi pianificati, vengono dunque demandate alle DAO, che si occupano anche della gestione delle transazioni e della connessione al database. Successivamente sono elencate le classi appartenenti a questo package ed evidenziati i metodi per effettuare operazioni di inserimento, visualizzazione e cancellazione per alcune di esse.

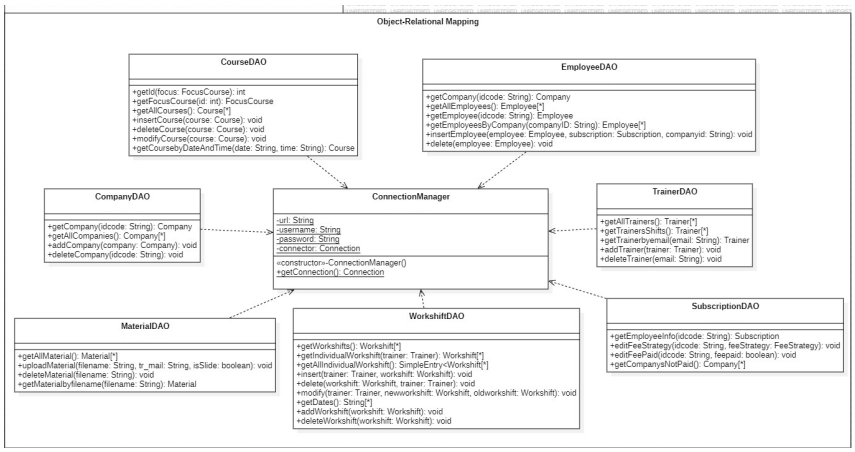

Figure 17: Class diagram del package ORM

3.2.3.1 ConnectionManager La classe *ConnectionManager* è responsabile della gestione della connessione a un database PostgreSQL utilizzando JDBC. Essa implementa il *Singleton Pattern* per garantire che esista una sola istanza della connessione durante l'esecuzione del programma. Il metodo *getConnection()* verifica se la connessione è già attiva e, in caso contrario, la crea utilizzando il driver JDBC. Questo approccio consente di ottimizzare l'uso delle risorse e ridurre il carico sul database, evitando la creazione ripetitiva di connessioni. Di seguito il codice che implementa ciò che è stato descritto:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionManager {
    private static final String url = "jdbc:postgresql://localhost:5433/myCourseAppdb";
    private static final String username = "postgres";
    private static final String password = "psw";
    private static Connection connector = null;

    private ConnectionManager(){}

    static public Connection getConnection() throws SQLException, ClassNotFoundException {
        Class.forName("org.postgresql.Driver");
        if (connector == null)
            connector = DriverManager.getConnection(url, username, password);

        return connector;
    }
}
```

Figure 18: ConnectionManager.java

3.2.3.2 SubscriptionDAO La classe *SubscriptionDAO* si occupa della gestione delle entità *Subscription*, fornendo metodi per la creazione, aggiornamento e recupero delle iscrizioni. Il metodo *getEmployeeInfo()* prende in input l'identificativo dell'impiegato e restituisce l'iscrizione corrispondente, caratterizzata da tre parametri: la durata dell'iscrizione, l'identificativo della strategia tariffaria (che può variare in base al numero di impiegati iscritti), e un valore booleano che indica se la quota è stata pagata. Il metodo *editFeeStrategy()* gestisce la modifica della strategia tariffaria, passando da *SingleEmployeeFee* a *MultipleEmployeeFee* quando vengono iscritti più dipendenti dallo stessa ditta. Per il monitoraggio dei pagamenti, il metodo *getCompanyNotPaid()* (figura : 19)restituisce l'elenco delle aziende con iscrizioni non ancora saldate. Una volta effettuato il pagamento, il metodo *editFeePaid()* aggiorna lo stato dell'iscrizione impostando il booleano a *true*.

```java
public ArrayList<Company> getCompanysNotPaid() throws SQLException, ClassNotFoundException {
    Connection con = ConnectionManager.getConnection();
    String sql = "SELECT DISTINCT companies.idcode, companies.email, companies.name, companies.address,
        companies.cellphone FROM companies, employees WHERE companies.idcode = employees.companyid AND employees.feepaid = false";
    PreparedStatement ps = con.prepareStatement(sql);
    ResultSet rs = ps.executeQuery();
    ArrayList<Company> companies = new ArrayList<Company>();
    while (rs.next()) {
        String idcode = rs.getString("idcode");
        String email = rs.getString("email");
        String name = rs.getString("name");
        String address = rs.getString("address");
        String cellphone = rs.getString("cellphone");
        Company company = new Company(idcode, email, name, address, cellphone);

        companies.add(company);
    }
    return companies;
}
```

Figure 19: getCompanyNotPaid()

3.2.3.3 WorkshiftDAO La classe *WorkshiftDAO* è responsabile della gestione delle entità *Workshift* e della tabella associativa *workshifts_trainer*, che mappa i turni di lavoro assegnati ai singoli formatori. Oltre ai metodi per l'inserimento e la rimozione di turni, la classe fornisce il metodo *modify()*, che consente di aggiornare un turno assegnato a un formatore sostituendolo con uno nuovo. Il metodo *getWorkshifts()* restituisce tutti i turni disponibili, mentre *getIndividualWorkshift()* recupera tutti i turni associati a un singolo formatore. Il metodo *getAllIndividualWorkshift()* produce una panoramica completa di tutti i formatori e dei rispettivi turni. Infine, il metodo *getDates()* restituisce l'elenco delle date relative al periodo di svolgimento dei corsi, utile per la pianificazione dei turni.

3.2.3.4 CompanyDAO La classe *CompanyDAO* gestisce la persistenza delle informazioni relative alle aziende. Oltre ai metodi canonici di inserimento e cancellazione, offre il metodo *getCompany()*, che consente di recuperare un'azienda dato il suo codice identificativo, e *getAllCompany()*, che restituisce un elenco completo di tutte le aziende presenti nel sistema.

3.2.3.5 TrainerDAO La classe *TrainerDAO* si occupa della gestione degli formatori. Include, oltre ai metodi base per l'inserimento e la cancellazione (figura: 20, il metodo *getTrainerShifts()*, che restituisce un'associazione tra formatori e i loro turni di lavoro. Il

metodo *getTrainerByEmail()* permette di cercare un formatore a partire dalla sua email, funzione utile nei processi di autenticazione o comunicazione, mentre *getAllTrainer()* restituisce l'elenco completo di tutti i formatori registrati.

```
public void deleteTrainer(String email) throws SQLException, ClassNotFoundException {
    Connection con = ConnectionManager.getConnection();
    String sql = "DELETE FROM trainers WHERE email = ?";
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setString(1, email);
    ps.executeUpdate();
}
```

Figure 20: deleteTrainer()

3.2.3.6 MaterialDAO La classe *MaterialDAO* gestisce la persistenza dei contenuti multimediali nel sistema. Il metodo *uploadMaterial()* riceve in ingresso un parametro *filename*, che rappresenta il percorso del file da caricare nel sistema, anziché un oggetto *Material* vero e proprio. Per il recupero di un contenuto multimediale specifico è disponibile il metodo *getMaterialbyfilename()*, che consente di ottenere l'oggetto *Material* associato a un determinato percorso. La classe include inoltre metodi standard per la gestione dei dati, come *deleteMaterial()*, per rimuovere un file dal sistema, e *getAllMedia()*, che restituisce una lista completa di tutti i media archiviati.

3.2.3.7 EmployeeDAO La classe *EmployeeDAO* è responsabile della gestione dei dati relativi ai dipendenti iscritti ai corsi di formazione. Oltre ai metodi canonici per l'inserimento e la rimozione, la classe offre il metodo *getAllEmployees()*, che restituisce una lista di tutti gli impiegati registrati. Il metodo *getEmployee()* consente di ottenere un oggetto *Employee* a partire dal suo codice identificativo. È inoltre disponibile il metodo *getEmployeesByCompany()*, che restituisce una lista di impiegati associati a una determinata azienda, prendendo in ingresso l'identificativo della stessa azienda. Infine, il metodo *getCompany()* permette di recuperare l'oggetto *Company* associato a un dipendente, dato il codice identificativo di quest'ultimo.
Nella Figura 21 possiamo vedere come il metodo *getEmployee()* consenta di recuperare un dipendente dal database fornendo il relativo codice identificativo. Dopo aver ottenuto una connessione tramite *ConnectionManager*, viene eseguita una query SQL che seleziona tutti i dati dell'impiegato corrispondente. Se la query restituisce un risultato, i valori vengono estratti dall'oggetto *ResultSet* e utilizzati per creare un'istanza di *Employee*. Inoltre, il metodo gestisce il tipo di strategia di pagamento associata al dipendente, scegliendo tra *MultipleEmployeesFee* e *SingleEmployeeFee* in base al valore dell'attributo *idstrategy*. Infine, l'oggetto *Subscription* viene istanziato con le informazioni estratte e associato al dipendente prima di restituire il risultato. Se nessun dipendente corrisponde al codice fornito, il metodo restituisce *null*.

```
public Employee getEmployee(String idcode) throws SQLException, ClassNotFoundException {
    Connection con = ConnectionManager.getConnection();
    String sql = "SELECT idcode, name, surname, age, role, meetings, idstrategy, feepaid  FROM employees WHERE idcode = ?";
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setString(1, idcode);
    ResultSet rs = ps.executeQuery();
    if (rs.next()) {
        String name = rs.getString("name");
        String surname = rs.getString("surname");
        int age = rs.getInt("age");
        String role = rs.getString("role");
        int meetings = rs.getInt("meetings");
        boolean feepaid = rs.getBoolean("feepaid");
        FeeStrategy feestrategy;
        if(rs.getInt("idstrategy") == 1){
            feestrategy = new MultipleEmployeesFee();
        }
        else{
            feestrategy = new SingleEmployeeFee();
        }
        Subscription subscription = new Subscription(meetings, null, feestrategy, feepaid);
        Employee employee = new Employee(idcode, name, surname, age, role);
        employee.setSubscription(subscription);
        return employee;
    }
    return null;
}
```

Figure 21: getEmployee()

3.2.3.8 CourseDAO La classe *CourseDAO* si occupa della gestione dei corsi di formazione pianificati. Oltre alle operazioni di inserimento, aggiornamento e cancellazione, fornisce il metodo *getAllCourses()*, che restituisce una lista di tutti i corsi disponibili nel sistema. Il metodo *getCourseByDateAndTime()* consente di recuperare un corso specifica fornendo in input una data e un orario: se esiste un'attività pianificata per quel momento, viene restituito l'oggetto corrispondente. La classe gestisce inoltre la tabella *focuscourse*, che contiene i possibili macro-argomenti su cui i corsi sono improntati. A tal fine, sono implementati i metodi *getId()*, che restituisce l'identificativo numerico associato a un oggetto *FocusCourse*, e *getFocusCourse()*, che permette di ottenere l'oggetto *FocusCourse* corrispondente a un dato identificativo numerico.
Sotto si mostra il metodo *insertCourse()* che ha il compito di inserire un nuovo corso nella base di dati. Per farlo, ottiene una connessione al database tramite la classe `ConnectionManager`, quindi prepara una query SQL parametrizzata per l'inserimento dei dati. La data del corso viene trasformata da una stringa in un oggetto `java.sql.Date`, mentre l'orario viene convertito in un `java.sql.Time`. Successivamente, vengono impostati i valori nella `PreparedStatement` e la query viene eseguita con *executeUpdate()*. Infine, il metodo chiude la dichiarazione per liberare risorse. Questa gestione garantisce una corretta memorizzazione dei dati nella tabella *courses*.

```
public void insertCourse (Course course) throws SQLException, ParseException, ClassNotFoundException {
    Connection con = ConnectionManager.getConnection();
    SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");

    String sql = "INSERT INTO courses ( date, time, description, focuscourse) VALUES (?, ?, ?, ?)";
    PreparedStatement ps = con.prepareStatement(sql);

    java.util.Date utilDate = format.parse(course.getDate());
    Date sqlDate = new Date(utilDate.getTime());
    LocalTime localTime = LocalTime.parse(course.getTime(), DateTimeFormatter.ofPattern("HH:mm:ss"));
    ps.setDate(1, sqlDate);
    ps.setTime(2, Time.valueOf(localTime));
    ps.setString(3, course.getDescription());
    ps.setInt(4, getId(course.getType()));
    ps.executeUpdate();
    ps.close();

}
```

Figure 22: insertCourse()

3.3 Database

Per la gestione della persistenza dei dati è stato adottato *PostgreSQL*, un sistema di gestione di basi di dati relazionali ad oggetti, open-source e gratuito, ampiamente utilizzato per la sua affidabilità, scalabilità e compatibilità con le tecnologie Java impiegate nell'applicazione.

Le figure 23 e 24 illustrano rispettivamente il diagramma *Entity-Relationship*, che descrive le entità del dominio e le loro relazioni, e lo schema logico del database, con le tabelle relazionali e i vincoli di integrità.

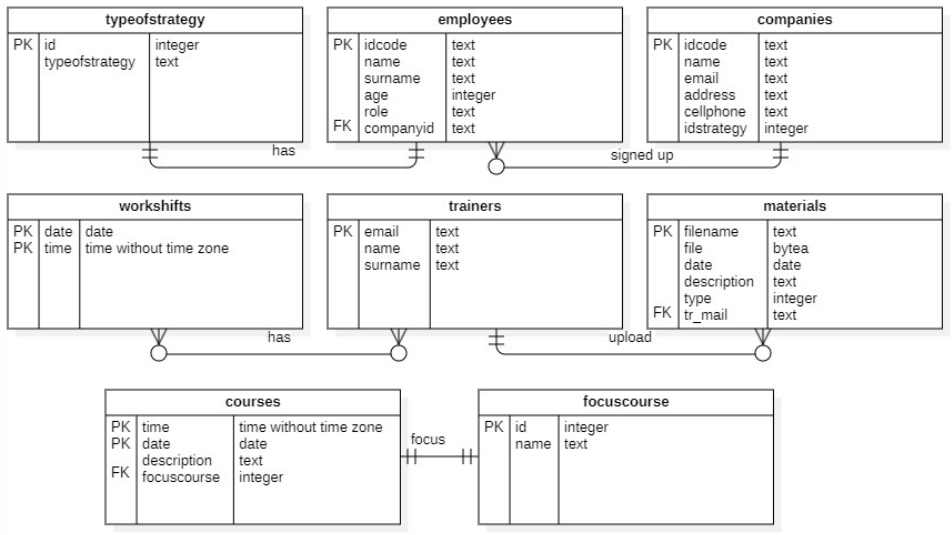

Figure 23: Diagramma Entity-Relationship del database

```
courses(PK(time,date),description, focuscourse),
FK(focuscourse) REF focuscourse

focuscourse(PK(id),name)

empoyees(PK(idcode),name, surname, age, role, companyid, meetings, idstrategy, feepaid)
FK(companyid)REF companies
FK(idstrategy)REF typeofstrategy

companies(PK(idcode), name, address, mail, cellphone)

typeofstrategy(PK(id), typeofstrategy)

materials(PK(filename), mediabytes, date, time, tr_mail),
FK(tr_mail) REF trainers

trainers(PK(email), name, surname)

workshifts(PK(date,time))

workshifts_trainers(PK(date,time,email),
FK(date,time) REF workshifts,
FK(email) REF trainers
```

Figure 24: Schema logico del database relazionale

4 Testing

Per la verifica del codice, è stato impiegato il framework JUnit, attraverso il quale sono stati sviluppati test per tutte le classi implementate, ad eccezione di quelle appartenenti al domain model, poiché contengono esclusivamente metodi di accesso ai dati, come getter e setter, che non necessitano di specifici test unitari. L'organizzazione dei test segue la struttura mostrata in figura 25, garantendo una copertura adeguata e un'efficace validazione delle funzionalità implementate.

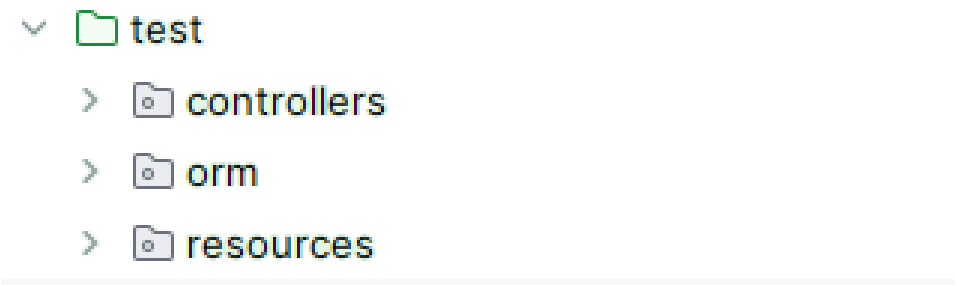

Figure 25: Organizzazione dei test

4.1 Test package Controllers

Il package `controllers` contiene le classi di test dedicate alla verifica delle funzionalità principali dei vari controller dell'applicazione, come amministratore, formatore e azienda. Questi test assicurano che le operazioni critiche come la gestione di materiali didattici, la consultazione di dati utente e l'interazione con il database siano implementate correttamente e funzionino come previsto in scenari realistici.

Per assicurare la coerenza dei test ed evitare interferenze tra i casi di prova, ciascun test è strutturato con blocchi `try-catch` per la gestione delle eccezioni. Inoltre, al termine di ogni test viene eseguita una fase di pulizia dei dati inseriti, assicurando così l'indipendenza e la ripetibilità dei test. Questo approccio contribuisce alla stabilità complessiva del sistema e alla qualità del software.

4.1.1 AdminTest

La classe `AdminTest` si occupa della verifica dei principali metodi della classe `Admin`, responsabile della gestione delle funzionalità amministrative del sistema, come la creazione, modifica, visualizzazione ed eliminazione dei corsi, nonché la gestione degli impiegati tramite l'uso dei rispettivi oggetti DAO, simulando così scenari d'uso concreti.

Il metodo *setCourse()* viene testato per verificare che i dati relativi a un nuovo corso vengano correttamente salvati nel database e che i valori persistiti siano conformi a quelli previsti. Con *modifyCourse()* si verifica la possibilità di aggiornare le informazioni di un corso esistente, assicurandosi che le modifiche vengano applicate senza perdita di dati. Il metodo *deleteCourse()* accerta la corretta eliminazione di un corso, verificando che non sia più presente nel sistema. Il metodo *viewCourses()* (figura: 26) permette invece di controllare la corretta restituzione della lista dei corsi, mentre *viewEmployeesList()* consente di validare l'estrazione della lista dei dipendenti registrati.

I test coprono l'intero ciclo di vita di un corso: inserimento, visualizzazione, modifica e cancellazione, oltre alla gestione degli impiegati. Ogni test verifica la corretta registrazione e manipolazione dei dati nel database, contribuendo a garantire l'affidabilità delle operazioni svolte dalla componente amministrativa del sistema.

```java
@Test
public void  viewCourses(){
    Course course = new Course("2025-12-12", "12:00:00", "test", FocusCourse.JAVA);
    CourseDAO courseDAO = new CourseDAO();
    Admin admin = new Admin();
    ArrayList<Course> courses;
    try{
        courseDAO.insertCourse(course);
        courses = admin.viewCourses();
        assertEquals( courses.get(courses.size()-1).getDate(), course.getDate());
        assertEquals( courses.get(courses.size()-1).getTime(), course.getTime());
        assertEquals( courses.get(courses.size()-1).getDescription(), course.getDescription());
        assertEquals( courses.get(courses.size()-1).getType(), course.getType());
    } catch (SQLException | ParseException | ClassNotFoundException e) {
        e.printStackTrace();
    } finally{
        try {
            courseDAO.deleteCourse(course);
        } catch (SQLException | ParseException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 26: viewCourses()

4.1.2 CompanyControllerTest

La classe `CompanyControllerTest` verifica il corretto funzionamento delle funzionalità offerte dalla classe `CompanyController`, responsabile della gestione operativa delle aziende, dei dipendenti, dei materiali didattici e dei corsi.

Il metodo *registerEmployee()* viene testato assicurandosi che l'inserimento di un nuovo dipendente in un'azienda comporti la corretta persistenza delle informazioni anagrafiche e contrattuali, come anche l'assegnazione automatica della strategia tariffaria più adeguata, in base al numero di dipendenti già associati all'azienda. Viene inoltre verificata la modifica della strategia tariffaria per i dipendenti esistenti quando viene superata la soglia prevista.

Il metodo che visualizza i materiali multimediali, *viewSlidesAndVideos()*, consente di testare il recupero dei file didattici presenti nel sistema. A tale scopo, un trainer fittizio carica un file di test che viene poi recuperato e validato, confermando la corretta associazione con l'utente e la corretta scrittura su file system.

La funzionalità di visualizzazione dei corsi offerti dal sistema viene testata tramite *viewCourses()*, assicurandosi che i dati dei corsi inseriti siano accessibili correttamente. Viene quindi inserito un corso di test e successivamente recuperato per verificare la coerenza delle informazioni.

Infine, con il metodo *viewEmployeeInfo()* viene verificata la corretta restituzione delle informazioni dei dipendenti associati a una determinata azienda. Il test accerta che tutti gli attributi anagrafici e contrattuali del dipendente siano correttamente persistiti e successivamente accessibili tramite il controller.

4.1.3 TrainerControllerTest

Rappresenta una suite di test dedicata alla verifica del comportamento della classe `TrainerController`. Essa include metodi per testare operazioni fondamentali eseguibili da un formatore, come il caricamento, la visualizzazione e la cancellazione di materiale didattico (slide e video), nonché la consultazione delle informazioni relative ai dipendenti, ai corsi e ai turni di lavoro. I test simulano l'interazione con il database tramite l'aggiunta e rimozione di dati temporanei e verificano la correttezza dei risultati attesi, utilizzando metodi di asserzione.

4.2 Test package ORM

Per garantire una corretta validazione delle operazioni sui dati, tutte le classi DAO implementano test unitari seguendo un ordine ben definito, stabilito mediante il framework JUnit e l'annotazione `@TestMethodOrder(MethodOrderer.OrderAnnotation.class)`. Questo approccio assicura che i test vengano eseguiti in una sequenza logica e coerente, evitando problemi legati alla dipendenza tra le operazioni.

In particolare, la struttura dei test prevede:

- **Test di inserimento**: verificano che i dati siano correttamente registrati nel database.
- **Test di recupero**: controllano che le informazioni precedentemente inserite possano essere lette e utilizzate correttamente.
- **Test di manipolazione**: assicurano che le operazioni di aggiornamento e modifica funzionino senza errori.
- **Test di eliminazione**: garantiscono che i dati possano essere rimossi senza compromettere l'integrità del database e mantenendo l'ambiente di test pulito.

L'adozione di questa metodologia ha permesso di migliorare la robustezza delle operazioni di accesso ai dati, riducendo il rischio di errori e garantendo maggiore affidabilità al sistema. Questo approccio è stato implementato grazie all'ausilio dell'Intelligenza Artificiale, che ha suggerito strategie ottimali per organizzare i test in modo più efficace.

4.2.1 TrainerDAOTest

Tra le varie classi, è stato scelto di approfondire *TrainerDAOTest* poiché rappresenta un esempio chiaro della metodologia utilizzata e mostra l'interazione tra più funzionalità legate alla gestione dei formatori.

Il primo test, *testAddTrainer*, si occupa di inserire un nuovo formatore, stabilendo una base dati da cui partire. I successivi test, *testGetTrainerByEmail*, *testGetAllTrainers* e *testGetTrainersShifts*, verificano la corretta registrazione e il recupero delle informazioni, assicurando che i dati siano accessibili in maniera coerente. L'ultimo test, *testDeleteTrainer*, completa il ciclo controllando che l'eliminazione del formatore dal database sia eseguita correttamente, permettendo di mantenere un ambiente di test pulito.

Questa classe riflette l'approccio sistematico adottato per tutte le DAO, garantendo affidabilità e coerenza nella gestione dei dati.

```java
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class TrainerDAOTest {

    private static TrainerDAO trainerDAO;

    private static final String TEST_EMAIL = "test.trainer@example.com";
    private static Trainer testTrainer;

    @BeforeAll
    public static void setup() {
        trainerDAO = new TrainerDAO();
        testTrainer = new Trainer(TEST_EMAIL, "TestName", "TestSurname");
    }

    @Test
    @Order(1)
    public void testAddTrainer() {
        try {
            trainerDAO.addTrainer(testTrainer);
        } catch (Exception e) {
            fail("Try to add a trainer failed: " + e.getMessage());
        }
    }

    @Test
    @Order(2)
    public void testGetTrainerByEmail() {
        try {
            Trainer t = trainerDAO.getTrainerbyemail(TEST_EMAIL);
            assertNotNull(t);
            assertEquals(TEST_EMAIL, t.getEmail());
            assertEquals(testTrainer.getName(), t.getName());
            assertEquals(testTrainer.getSurname(), t.getSurname());
        } catch (Exception e) {
            fail("Recupero trainer fallito: " + e.getMessage());
        }
    }

    @Test
    @Order(3)
    public void testGetAllTrainers() {
        try {
            ArrayList<Trainer> trainers = trainerDAO.getAllTrainers();
            assertNotNull(trainers);
            assertTrue(trainers.stream().anyMatch(t -> t.getEmail().equals(TEST_EMAI
        } catch (Exception e) {
            fail("Recupero lista trainers fallito: " + e.getMessage());
        }
    }

    @Test
    @Order(4)
    public void testGetTrainersShifts() {
        try {
            ArrayList<Trainer> trainers = trainerDAO.getTrainersShifts();
            assertNotNull(trainers);
            // Il trainer inserito dovrebbe comparire nella lista, anche se non ha t
            assertTrue(trainers.stream().anyMatch(t -> t.getEmail().equals(TEST_EMAI
            // Per ogni trainer i workshifts non devono essere null (possono essere
            trainers.forEach(t -> assertNotNull(t.getWorkshifts()));
        } catch (Exception e) {
            fail("Recupero trainers con turni fallito: " + e.getMessage());
        }
    }

    @Test
    @Order(5)
    public void testDeleteTrainer() {
        try {
            trainerDAO.deleteTrainer(TEST_EMAIL);
            Trainer t = trainerDAO.getTrainerbyemail(TEST_EMAIL);
            assertNull(t);
        } catch (Exception e) {
            fail("Cancellazione trainer fallita: " + e.getMessage());
        }
    }
```

4.3 Risultati dei test

L'esecuzione dei test sui package Orm e Controllers ha confermato la correttezza e la stabilità delle operazioni implementate. Tutti i 51 test sono stati completati con successo, garantendo la validità delle funzionalità di gestione dei dati e della logica applicativa. Questo risultato evidenzia la solidità del codice, la coerenza tra i moduli e l'adeguata implementazione delle logiche di accesso e manipolazione delle informazioni. Di seguito vengono riportati i dettagli delle verifiche effettuate e le rispettive conferme di buon funzionamento.

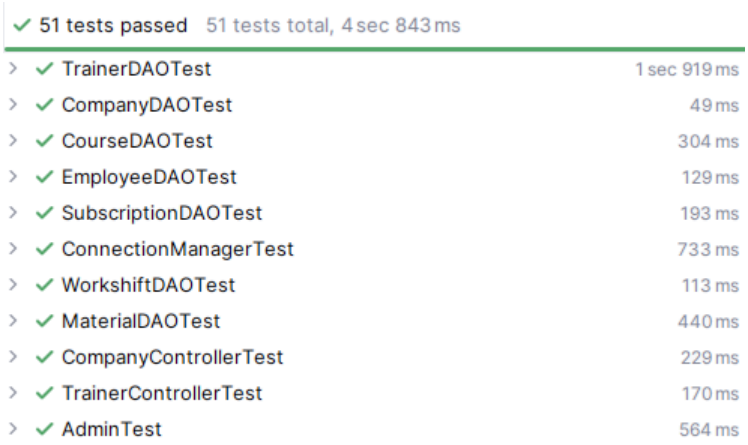

Figure 27: Risultati dei test effettuati

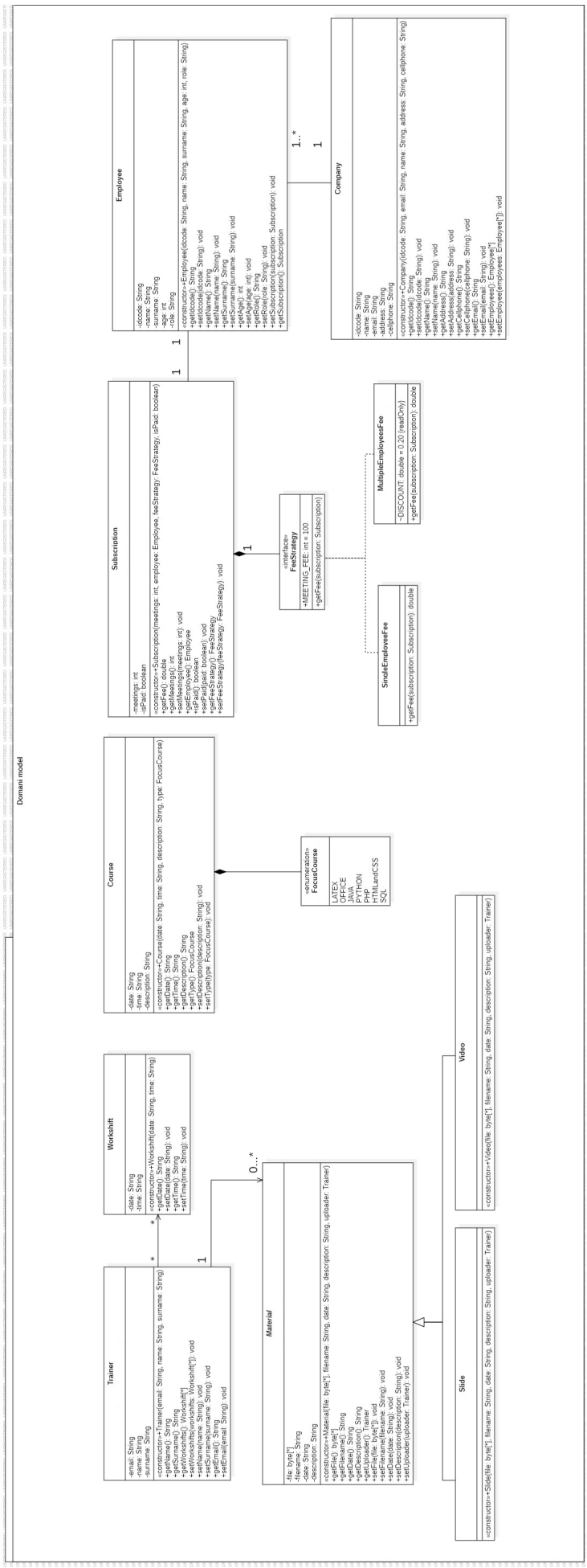

Figure 28: Class diagram del Domain Model in formato orizzontale