Ingegneria del Software
Corso di Laurea in Ingegneria Informatica
Univeristà degli Studi di Firenze

Shared Calendar Application

Alessio Bonacchi, Marco Guerra
Sessione in Aprile
2021

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Indice

Introduzione

Motivazione

Abbiamo deciso di implementare un'applicazione per gestire calendari condivisi perché curiosi del suo funzionamento. Ad oggi ne esistono diverse sul mercato, anche piuttosto popolari, ma si prospetta sempre la necessità di implementare calendari personalizzati per diverse tipologie di business, che si integrino anche eventualmente con il gestionale d'azienda.

Nella nostra scelta c'è anche l'intenzione di mettere a frutto gli studi intrapresi sia nel Corso di Ingegneria del Software che in quello di Basi di Dati.

Statement

La nostra applicazione ha l'intento di realizzare un sistema di calendari condivisi, sicuro ed efficiente. Ogni calendario ha un proprietario e può essere condiviso con altri utenti, i quali possono modificarlo secondo le autorizzazioni a loro riconosciute.
Gli utenti possono accedere al proprio calendario attraverso un sistema di autenticazione. L'interfaccia di gestione del calendario offre all'utente la possibilità di creare nuovi eventi, modificarli e cancellarli.
Ogni evento è caratterizzato da diversi attributi, quali nome, descrizione, luogo e data.
L'interazione con il calendario da parte dell'utente è realizzata tramite un'interfaccia grafica che permette all'utente di visualizzare gli eventi in base al giorno, settimana o mese.
E' possibile ricercare l'evento per nome e luogo, tra gli eventi passati e quelli programmati.
E' stato implementato un sistema di notifiche che avvisa l'utente degli eventi programmati per il giorno stesso.
I calendari sono condivisi con la seguente modalità: un utente può scegliere di condividere con altri un calendario di cui è proprietario.

Metodo

L'applicazione è sviluppata in Java.

Il database sul quale vengono archiviati i dati degli utenti e dei calendari è gestito tramite PostgreSQL.

La connessione al database è realizzata tramite le librerie di Java DataBase Connectivity (JDBC).

I test vengono realizzati tramite le librerie di JUnit.

La grafica è realizzata tramite Java Swing. In particolare abbiamo fatto uso dei componenti offerti da Mindfusion[1] per quanto riguarda la visualizzazione degli eventi.

Per i diagrammi di classi e pattern si utilizza lo standard UML.

1. Progettazione

1.1 Use Case Diagram

Sono stati individuati i casi d'uso in Figura 1.1, al fine di definire un piano di progettazione chiaro ed esaustivo. Inoltre sono specificati in dettaglio (template) i due Use Case che stanno all base del funzionamento dell'applicazione.

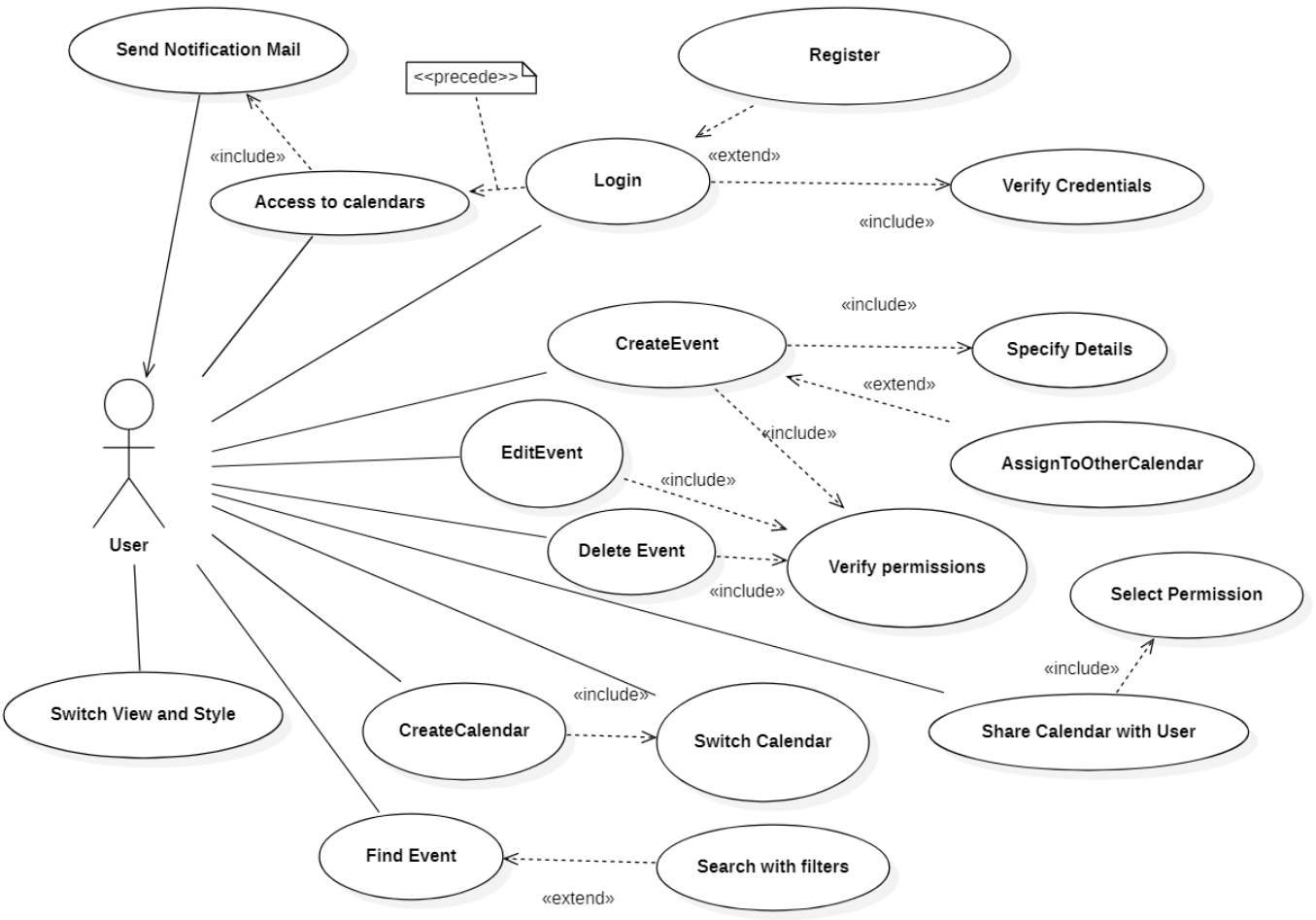

Figura 1.1: Casi d'uso - diagramma

UC	LOGIN
Level	User Goal
Actor	User
Basic Course	1-L'utente riempie i campi dello username e della password 2-L'utente preme sul bottone "Login" per accedere 3-Il sistema verifica che le credenziali di accesso sono valide 4-L'utente può accedere ad i suoi calendari
Alternative Course	3a-Il sistema non riesce a reperire le credenziali di accesso perchè queste non sono valide 3a-Il sistema mostra un dialog di fallimento 3b-Il sistema verifica che i campi di username e password sono vuoti 3b- Il sistema mostra un dialog di fallimento

UC	CREATE EVENT
Level	User Goal
Actor	User
Basic Course	1-L'utente passa il cursore sul menu "File" 2-L'utente clicca sull' elemento di menu "Create Event" 3-Il sistema mostra la vista "EditEventView" 4-L'utente seleziona la data dal calendario integrato nella vista 5-L'utente digita il nome dell'evento nell'apposito campo 6- L'utente digita la location dell'evento nell'apposito campo 7- L'utente digita la descrizione dell'evento nell'apposito campo 8- L'utente seleziona l'ora di inzio e fine dell'evento dagli appositi campi 9-L'utente clicca su "Edit" per creare l'evento 10-L'evento viene creato e mostrato nel calendario
Alternative Course	10a L'evento non viene creato perchè non sono stati specificati dei dati essenziali 10a- Il sistema mostra un dialog di fallimento 10b- L'evento non viene creato perchè si è verificata un'inconsistenza nei dati inseriti 10b- Il sistema mostra un dialog di fallimento

1.2 Class Diagram

Sono stati individuati diversi package in cui dividere il diagramma UML delle classi:

- Model: contiene la business logic dell'applicazione
- Controller: gestisce l'interoperabilità tra Model e View
- View: definisce le viste della GUI (Figura 1.4)
- Utils: contiene classi di supporto al funzionamento dell'applicazione

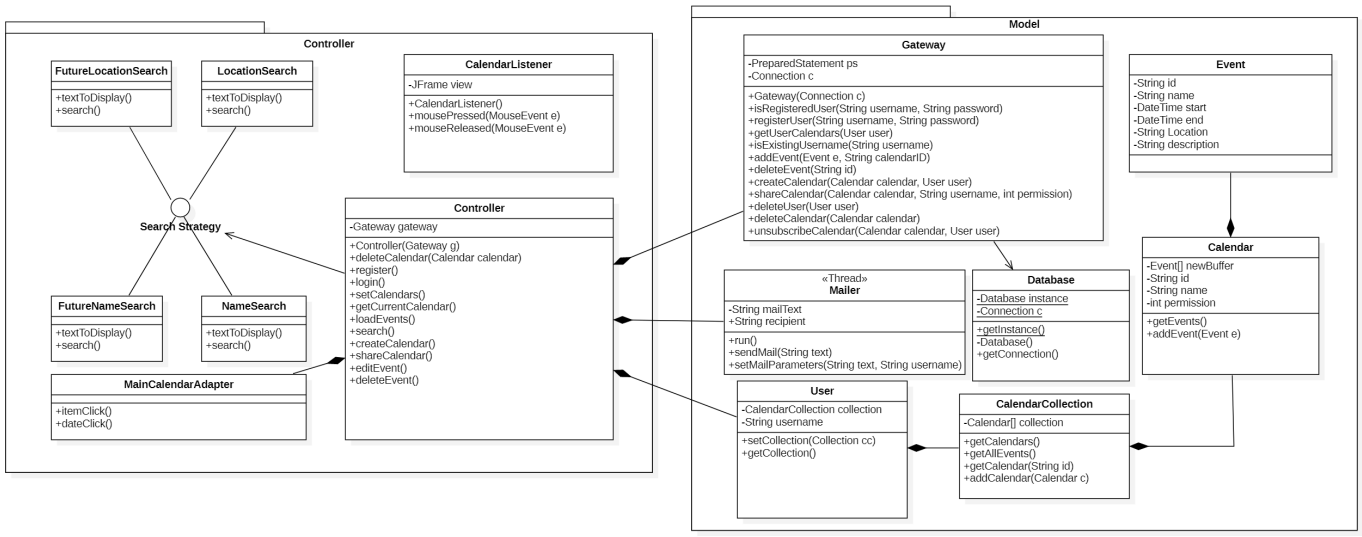

Figura 1.2: UML - class diagram

1.3 Entity Relationship Diagram

Al fine di progettare un database che soddisfi i requisiti di normalizzazione di terza forma, abbiamo individuato relazioni ed entità tramite un diagramma. Le due relazioni indicate sono realizzate tramite le tabelle 'Participation' (che presenta un attributo 'type' per la gestione di permessi e autorizzazioni) e 'CalendarEvents'.

La tabella 'Login' contiene gli utenti con le credenziali, la tabella 'Calendar' i calendari sia condivisi che non, e la tabella 'Events' contiene tutti gli eventi.

La condivisione si realizza effettivamente aggiungendo un record in 'Participation' dove compaiono il calendario condiviso e l'utente con cui si condivide, specificando il ruolo del nuovo utente.

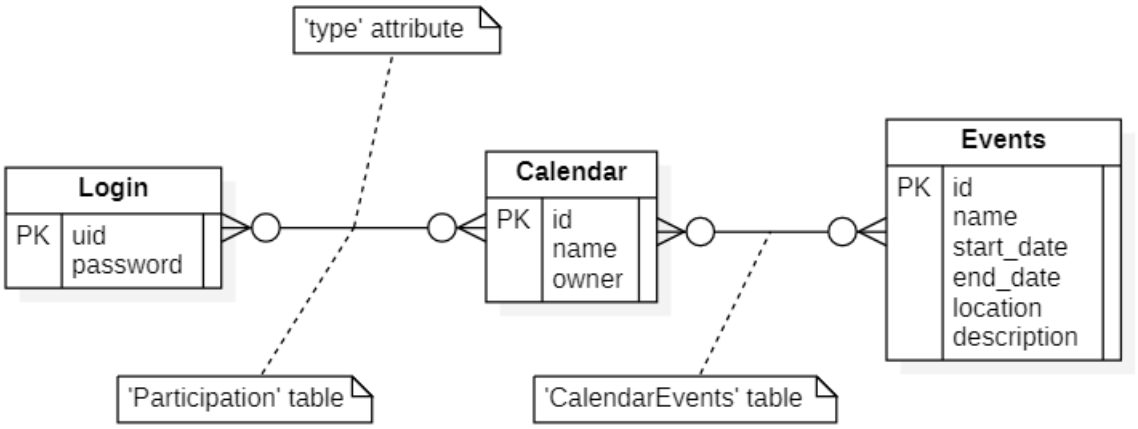

Figura 1.3: Entity Relationship Diagram

1.4 Window Navigation Diagram

Di seguito si riporta come le varie finestre nell'interfaccia grafica siano navigabili. La visualizzazione delle singole pagine è demandata all'operazione applicata, come specificato in figura.

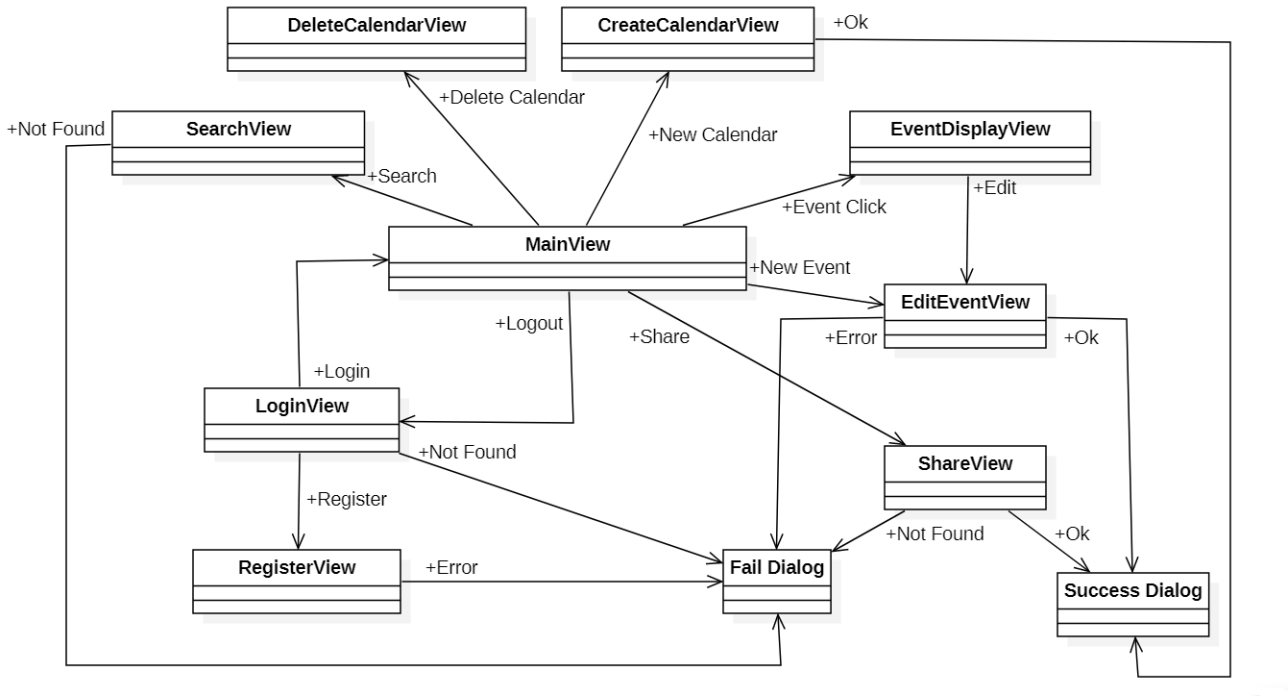

Figura 1.4: Page Navigation Diagram

2. Implementazione

Partendo dal Class Diagram e dal Use Case Diagram, si procede implementando le classi. In particolare si presta attenzione a disaccoppiare l'implementazione della GUI , realizzata tramite istanze di JFrame di Java Swing, e del modello, che può accedere al Database tramite JDBC e Query SQL.

2.1 Classi e Interfacce

Il nostro intento è di sviluppare un'applicazione che realizzi sostanzialmente un calendario virtuale. I calendari sono composti di eventi, e appartengono a degli utenti che possono condividerli tra di loro. Date queste specifiche, è necessario che, utilizzando la programmazione ad oggetti, ogni entità abbia la sua rappresentazione in forma di classe Java. E' indispensabile inoltre che per ogni classe vengano definite le responsabilità e le finalità specifiche.

2.1.1 User

La classe User rappresenta il modello di un utente.

Un utente infatti può essere caratterizzato da diversi attributi, come il suo username o la sua collezione dei calendari.

La classe User ci permette quindi di mettere in relazione le informazioni di un utente, una volta effettuato l'accesso all'applicazione.

2.1.2 Calendar

La classe Calendar rappresenta il modello di un calendario.

Un calendario ci permette di considerare un insieme di eventi, in modo da aggregarli per categoria.

In questo modo, Calendar ci permette di definire attributi per una collezione di eventi, come i permessi che l'utente ha su di questi, come sola lettura o scrittura.

2.1.3 CalendarCollection

La classe CalendarCollection rappresenta il modello di una lista di calendari.

Una CalendarCollection si rende necessaria nel momento in cui si vogliono caricare in memoria tutti i calendari di un utente, di modo da averli disponibili per essere visualizzati e modificati.

In questo modo, CalendarCollection definisce un insieme di calendari, per poterli considerare in gruppo oppure singolarmente.

2.1.4 Event

La classe Event rappresenta il modello di un evento.
Gli eventi sono alla base di una applicazione di calendari condivisi.

La rappresentazione di un evento comprende molti attributi di diverso tipo, quali il titolo, la durata temporale, il luogo e la descrizione. Per ogni attributo è necessario definire dei metodi 'getter' e 'setter'.

2.1.5 Database

La classe Database è un Singleton (Figura 2.1) che ci permette di istanziare la connessione con il database, gestito con PostgreSQL dall'applicazione.

Ha la responsabilità di disaccoppiare la creazione e il mantenimento di una connessione dalle funzionalità dell'applicazione e dalle operazioni che possono essere effettuate (Query).

2.2 Desisng Patterns

La programmazione a oggetti è caratterizzata da problemi di incapsulamento e divisione delle responsabilità. Nel Corso di Ingegneria del Software abbiamo imparato a risolvere questi problemi ricorrenti tramite l'implementazione di Design Patterns. Si specificano di seguito quali Desing Patterns sono stati adottati e in che modo.

2.2.1 Singleton

Il Singleton Pattern permette la creazione di una sola istanza di un certo oggetto, e la rende accessibile.

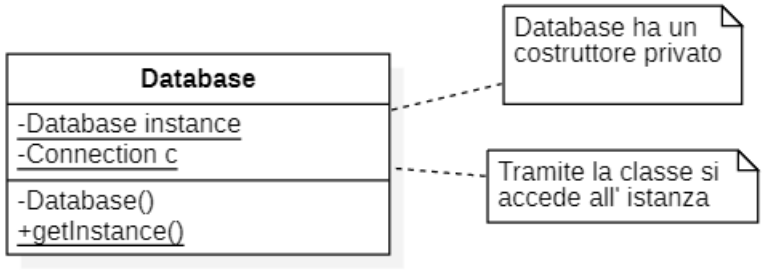

Figura 2.1: Singleton - design pattern

Questo è un pattern creazionale indispensabile in quelle situazioni in cui non ha senso l'istanza di più oggetti di una stessa classe, o potrebbe persino interferire con il funzionamento dell'applicazione.

Il Singleton ha la responsabilità di creare e rendere accessibile la sola istanza tramite un riferimento statico.

Nella nostra applicazione il Singleton ci ha permesso di assicurarsi che soltanto una connessione al Database venga gestita dall'applicazione, in modo che non si verifichino ambiguità nel suo utilizzo.

2.2.2 Builder

Il Builder Pattern è costituito da una classe che permette di selezionare le specifiche per la creazione di un oggetto.

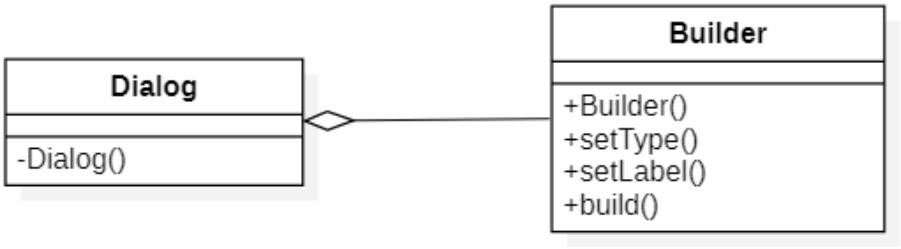

Figura 2.2: Builder - design pattern

Se ci sono degli oggetti che differiscono tra loro soltanto per specifici attributi, il Builder può semplificare la produzione di istanze di questi. In particolare la responsabilità del Builder è quella di disaccoppiare la costruzione di un oggetto complesso dalla sua rappresentazione.

Abbiamo utilizzato un Builder nella nostra applicazione per realizzare Dialogs che comunichino all'utente l'esito delle operazioni che compie sul modello. In particolare si rende più comprensibile la creazione di un Dialog e risolve il problema della ridondanza del codice.

2.2.3 Strategy

L'implementazione dello Strategy Pattern nella nostra applicazione è mostrata in figura:

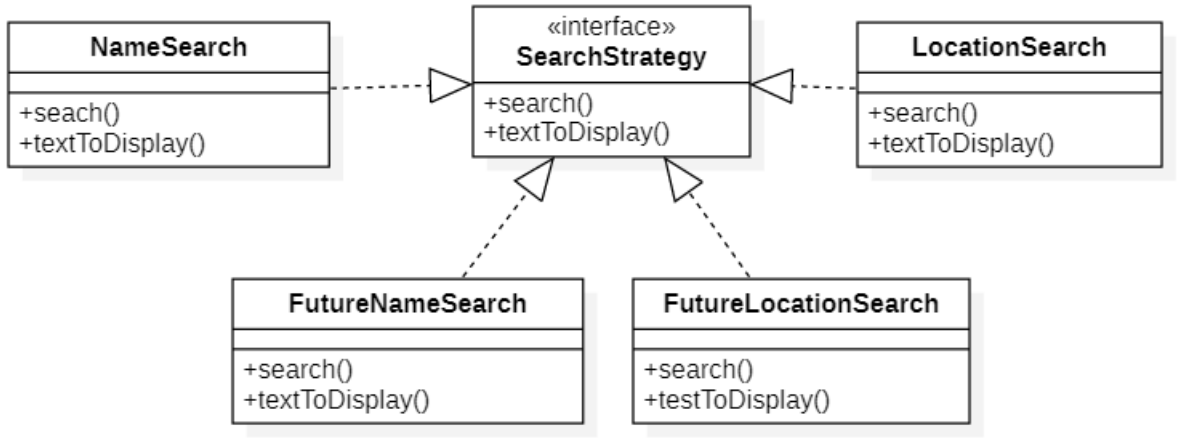

Figura 2.3: Strategy - design pattern

Lo Strategy Pattern ci permette di definire metodologie multiple per la ricerca degli eventi. In prima battuta è utile in quanto la nostra applicazione necessita di più varianti di un algoritmo, permettendoci così di definire quattro implementazioni dell'algoritmo di ricerca.

In secondo luogo ci permette di scegliere in modo dinamico che tipo di comportamento realizzare al momento che l'utente scelga la modalità di ricerca.

Nel nostro caso abbiamo implementato 4 modi per ricercare gli eventi:

- NameSearch: ricerca per nome anche eventi passati.
- FutureNameSearch: ricerca per nome dell'evento.
- LocationSearch: ricerca per location anche eventi passati.
- FutureLocationSearch: ricerca per location dell'evento.

2.2.4 Gateway

Il pattern Gateway prevede una classe 'Gateway' che traduce le chiamate specifiche del client in un protocollo compatibile con un servizio esterno.

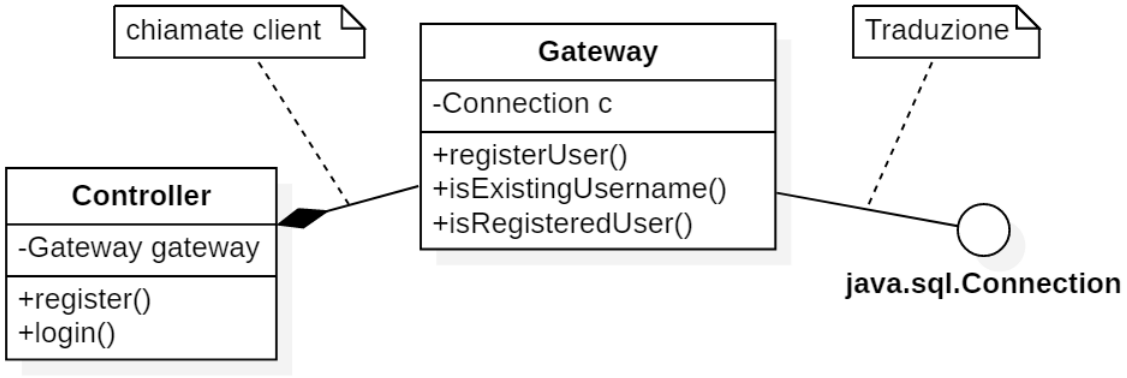

Figura 2.4: Gateway - design pattern

Il Gateway risulta particolarmente efficace nel nostro caso specifico: se fosse necessario modificare la struttura di una query, sarebbe sufficiente agire sui metodi del Gateway, piuttosto che in ogni sezione del Controller che ne fa uso.

La responsabilità del Gateway sta proprio in questo: disaccoppiare il client dall'interfaccia API del database.

Nella nostra applicazione, il Gateway traduce le azioni dell'utente in Query SQL.

2.2.5 RBAC

Il Role Based Access Control (RBAC[2]) Pattern prevede una classe la cui responsabilità è autorizzare le oeprazioni di un agente su un certo oggetto, in base a dei ruoli ben definiti.

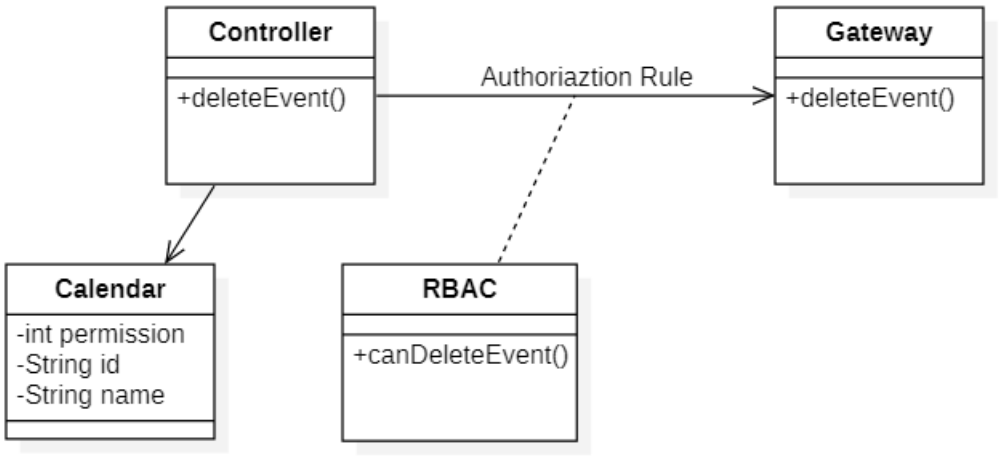

Figura 2.5: Role Based Access Control - design pattern

Il RBAC è indispensabile in fase di progettazione di un sistema con permessi e autorizzazioni in base ai ruoli: se infatti si necessita di una restrutturazione di permessi e ruoli, dovremo agire soltanto sulla singola classe.

La responsabilità del RBAC è quindi di disaccoppiare il sistema di controllo dell'accesso dal resto dell'applicazione.

Nel nostro caso specifico il RBAC controlla l'autorizzazione dell'utente ad agire su un certo calendario.

2.2.6 MVC

Il Model View Controller (MVC) è un pattern strutturale che divide il modello (Model) dell'applicazione dalla sua interfaccia grafica (View), e le gestisce attraverso un sistema di controllo (Controller).

Quando si progetta un' applicazione dotata di un' interfaccia grafica piuttosto rilevante, MVC è di fatto il metodo più efficace: si può infatti agire sulla grafica senza toccare il modello e viceversa, rendendoli comunque interoperabili.

La responsabilità di MVC infatti è proprio quella di disaccoppiare il modello dalla vista grafica, permettendo l'interazione soltanto tramite il controller.

Nel nostro caso, il modello è un insieme di classi che comprende Event, Calendar, User, Database, Gateway, mentre la grafica si articola in un insieme di JFrame navigabili.

2.3 Ulteriori Dettagli Implementativi

2.3.1 Mailer

La classe Mailer ha la responsabilità di gestire l'invio di e-mail tramite protocollo SMTP. La libreria da cui importiamo le funzionalità richieste è 'javax.mail'. In particolare il Mailer si occupa di inviare mail personalizzate in base agli eventi giornalieri dell'utente.

E' previsto che il nome utente sia un indirizzo mail valido: il controllo viene effettuate tramite Regular Expressions (RegEx) al momento della registrazione.

2.3.2 Edit Distance

Al fine di realizzare la metodologia di ricerca degli eventi all'interno della nostra applicazione abbiamo utilizzato l'algoritmo Edit Distance: viene specificata la distanza massima ammissibile tra chiave e target al fine di restituire solo risultati potenzialmente inerenti.

In particolare abbiamo definito quattro metodi per realizzare una ricerca filtrata sugli eventi utilizzando l'algoritmo di Edit Distance, specificati in Figura 2.3.

3. Testing

Questa sezione è dedicata alla documentazione dei test.

Per certificare il corretto funzionamento dell'applicazione, abbiamo implementato diversi test tramite la piattaforma di Testing JUnit. In particolare ci si vuole assicurare che l'accesso al database e l'esecuzione delle Query siano corrette e resilienti.

I metodi test sono definiti da @Test. Per ogni classe di test si include anche dei metodi di setup e pulizia, che si occupano di adeguare la configurazione del database all'esecuzione dei test.

3.1 Integration Testing

In questa sezione si documentano i test di tipo Integration Testing. In particolare si verifica il funzionamento dell'interfaccia grafica.

3.1.1 LoginTest

Questo test, di tipo Black-Box, asserisce il corretto funzionamento dell'interfaccia grafica per il login e la registrazione dell'utente.

Listing 3.1: Test login

```
@BeforeClass
public static void  init() throws AWTException {

    Database db = Database.getInstance();
    gateway = new Gateway(db.getConnection());
    robot = new Robot();

}

@Before
public void setUp(){ controller = new Controller(gateway); }

@AfterClass
public static void closeAll(){ controller.getMwView().close(); }
```

```
@Test
public void testUnsuccesfulLogin(){

    int x = controller.getCwView().getX()+41;
    int y = controller.getCwView().getY()+100;
    robot.mouseMove(x, y);
    controller.getLogView().getTextPassword().setText("wrong");
    controller.getLogView().getTextUser().setText("wrong");
    robot.mousePress(InputEvent.BUTTON1_DOWN_MASK);
    robot.mouseRelease(InputEvent.BUTTON1_DOWN_MASK);
    assertNull(controller.getMwView());

}

@Test
public void testSuccessfullLogin(){

    int x = controller.getCwView().getX()+40;
    int y = controller.getCwView().getY()+100;
    robot.mouseMove(x, y);
    controller.getLogView().getTextPassword().setText("a");
    controller.getLogView().getTextUser().setText("a");
    robot.mousePress(InputEvent.BUTTON1_DOWN_MASK);
    robot.mouseRelease(InputEvent.BUTTON1_DOWN_MASK);
    assertNotNull(controller.getMwView());

}
```

Nello specifico, il test verifica il comportamento dell'interfaccia di login tramite credenziali giuste ed errate.

3.2 Unit Testing

In questa sezione si documentano i test di tipo Unit Testing. In particolare si verifica il funzionamento della Business Logic.

3.2.1 EventDatabaseTest

Questo test, di tipo Gray-Box, ci permette di testare le operazioni di creazione e cancellazione degli eventi dal database.

Listing 3.2: Test event creation

```
@BeforeClass
public static void init(){

    Database db = Database.getInstance();
    gateway = new Gateway(db.getConnection());
    username1 = "username";
```

```
        calendar = new Calendar("test","CID", RBAC.getCreatorPermission());
        user = new User(username1);
        event = new Event("a","concerto",start_date,end_date,"casa","wow");
        gateway.createCalendar(calendar,user);

    }

    @Test
    public void checkCreateCalendar(){

        calendarCollection = gateway.getUserCalendars(user);
        assertNotNull(calendarCollection.getCalendar("test"));

    }

    @Test
    public void eventCreation(){

        calendarCollection = gateway.getUserCalendars(user);
        int sizeBefore = calendarCollection.getAllEvents().size();
        gateway.addEvent(event,calendar.getId());
        calendarCollection = gateway.getUserCalendars(user);
        int sizeAfter = calendarCollection.getAllEvents().size();
        assertTrue(sizeAfter==sizeBefore+1);
        gateway.deleteEvent(event.getId());

    }

    @Test
    public void deleteEvent(){

        calendarCollection = gateway.getUserCalendars(user);
        int sizeBefore = calendarCollection.getAllEvents().size();
        gateway.addEvent(event,calendar.getId());
        gateway.deleteEvent(event.getId());
        calendarCollection = gateway.getUserCalendars(user);
        int sizeAfter = calendarCollection.getAllEvents().size();
        assertTrue(sizeAfter==sizeBefore);

    }
```

Le manipolazioni degli eventi sono operazioni che stanno alla base del funzionamento dell'applicazione: è quindi naturale testare il loro funzionamento.

3.2.2 LoginDatabaseTest

Questo test, di tipo Gray-Box, ci permette di testare le operazioni di registrazione e login degli degli utenti al database.

Listing 3.3: Test User login and registration

```
@BeforeClass
  public static void init(){

      Database db = Database.getInstance();
      gateway = new Gateway(db.getConnection());
      sample = "prova";

  }

  @Before
  public void setUp(){

      sample = "prova";
      gateway.registerUser(sample,sample);

  }

  @After
  public void closeUp(){ gateway.deleteUser(user);}


  @Test
  public void registerNewUser(){

      assertTrue(gateway.isRegisteredUser(sample,sample));

  }

  @Test
  public void isExistingUsernameTest(){

      assertTrue(gateway.isExistingUsername(sample));

  }

  @Test
  public void checkDelete(){

      gateway.deleteUser(user);
      assertFalse(gateway.isRegisteredUser(sample,sample));

  }
```

E' necessario testare il corretto funzionamento della creazione di utenti anche per verficcare che il databse sia configurato in modo compatibile con l'applicazione.

3.2.3 ShareDatabaseTest

Questo test, di tipo Gray-box, ci permette di testare le operazioni di condivisione dei calendari con diversi permessi.

Listing 3.4: Test database sharing

```
@BeforeClass
public static void init(){

    Database db = Database.getInstance();
    gateway = new Gateway(db.getConnection());
    username1 = "Marco";
    user1 = new User(username1);
    calendar = new Calendar("test","CID", RBAC.getCreatorPermission());
    member = new User("Username");
    gateway.createCalendar(calendar,user);

}

@AfterClass
public static void reset(){ gateway.deleteCalendar(calendar); }

@After
public void setUp(){ gateway.unsubscribeCalendar(calendar,member); }

@Test
public void shareAsCreator(){

    gateway.shareCalendar(calendar,member.getUsername(), RBAC.getCreatorPermission());
    CalendarCollection cc1 = gateway.getUserCalendars(member);
    assertEquals(cc1.getCalendar(calendar.getId()).getPermission(),0);

}

@Test
public void shareAsOwner(){

    gateway.shareCalendar(calendar,member.getUsername(), RBAC.getOwnerPermission());
    CalendarCollection cc1 = gateway.getUserCalendars(member);
    assertEquals(cc1.getCalendar(calendar.getId()).getPermission(),1);
}

@Test
public void shareAsUser(){

    gateway.shareCalendar(calendar,member.getUsername(), RBAC.getUserPermission());
    CalendarCollection cc1 = gateway.getUserCalendars(member);
    assertEquals(cc1.getCalendar(calendar.getId()).getPermission(),2);

}
```

Poiché la condivisione dei calendari è l'oggetto di studio della applicazione sviluppata, è fondamentale garantirne il corretto funzionamento.

4. Demo

Seguono alcune immagini che illustrano la creazione di un calendario 'Progetto SWE' da parte dell'utente 'guerra@unifi.it', che decide di condividerlo con l'utente 'alessio@unifi.it con il ruolo di 'utente':

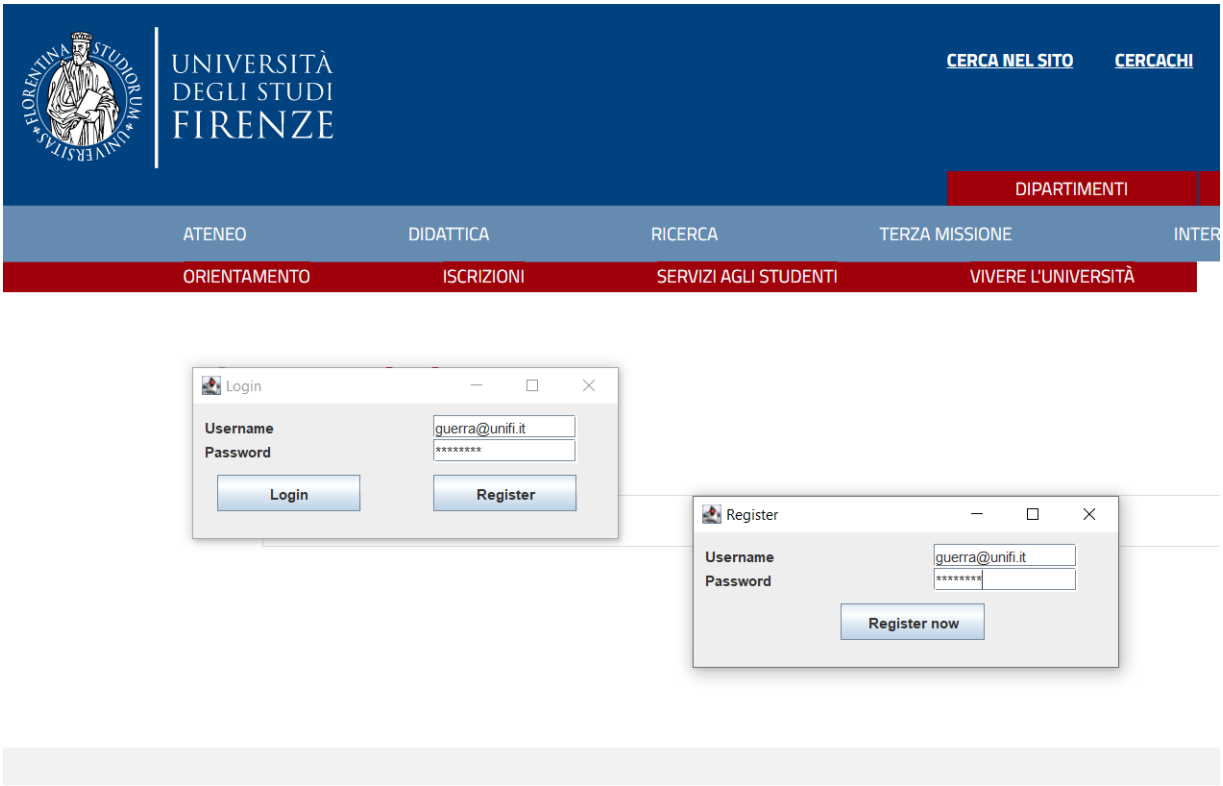

Figura 4.1: Demo - registrazione e login

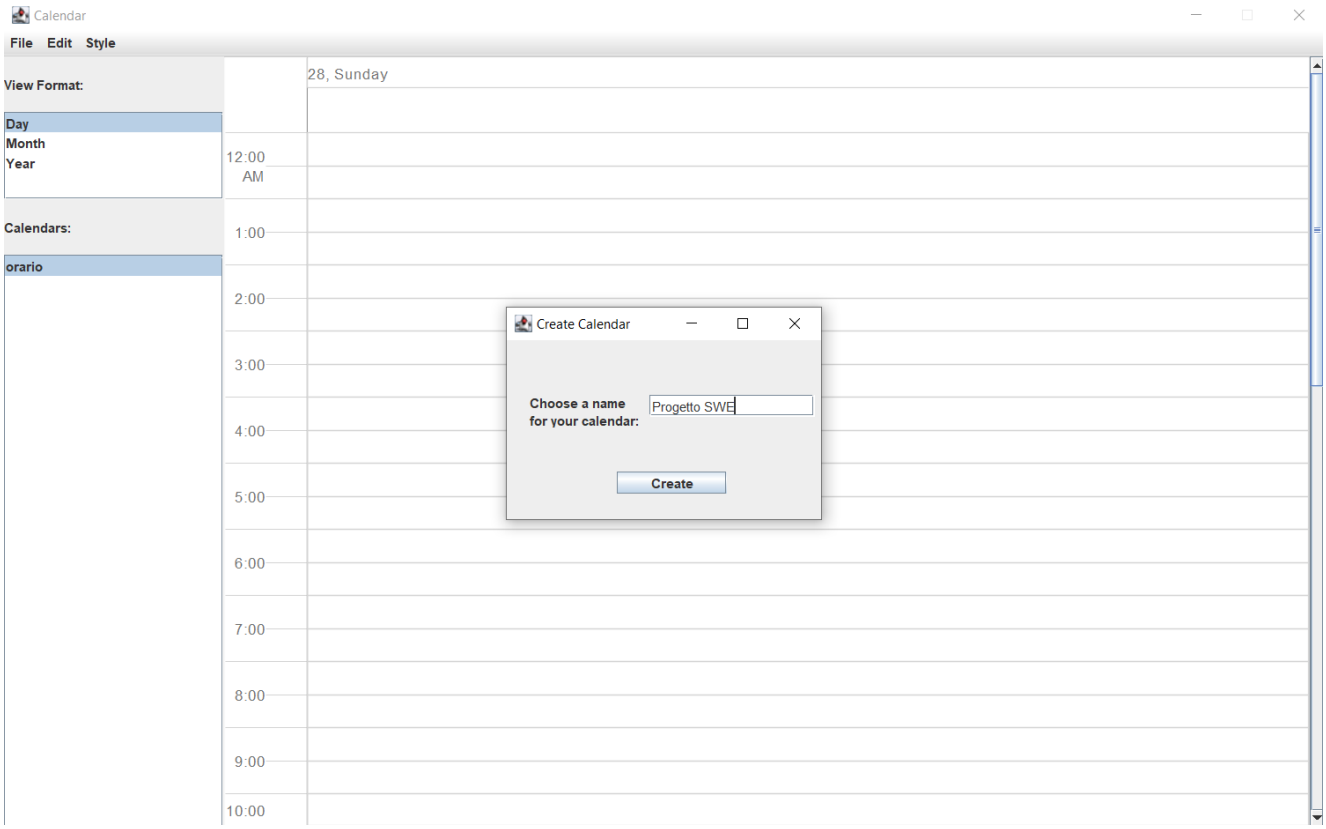

Figura 4.2: Demo - creazione calendario

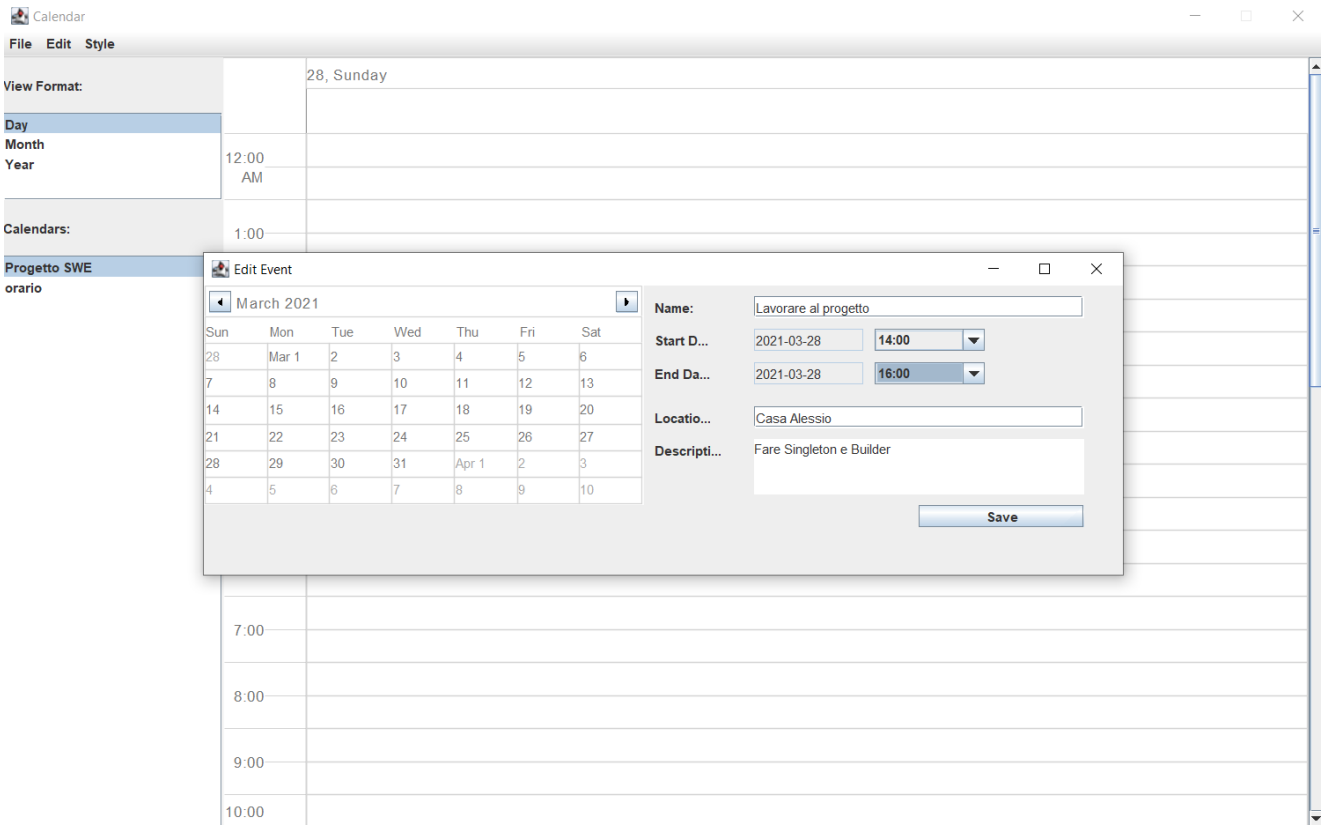

Figura 4.3: Demo - creazione evento

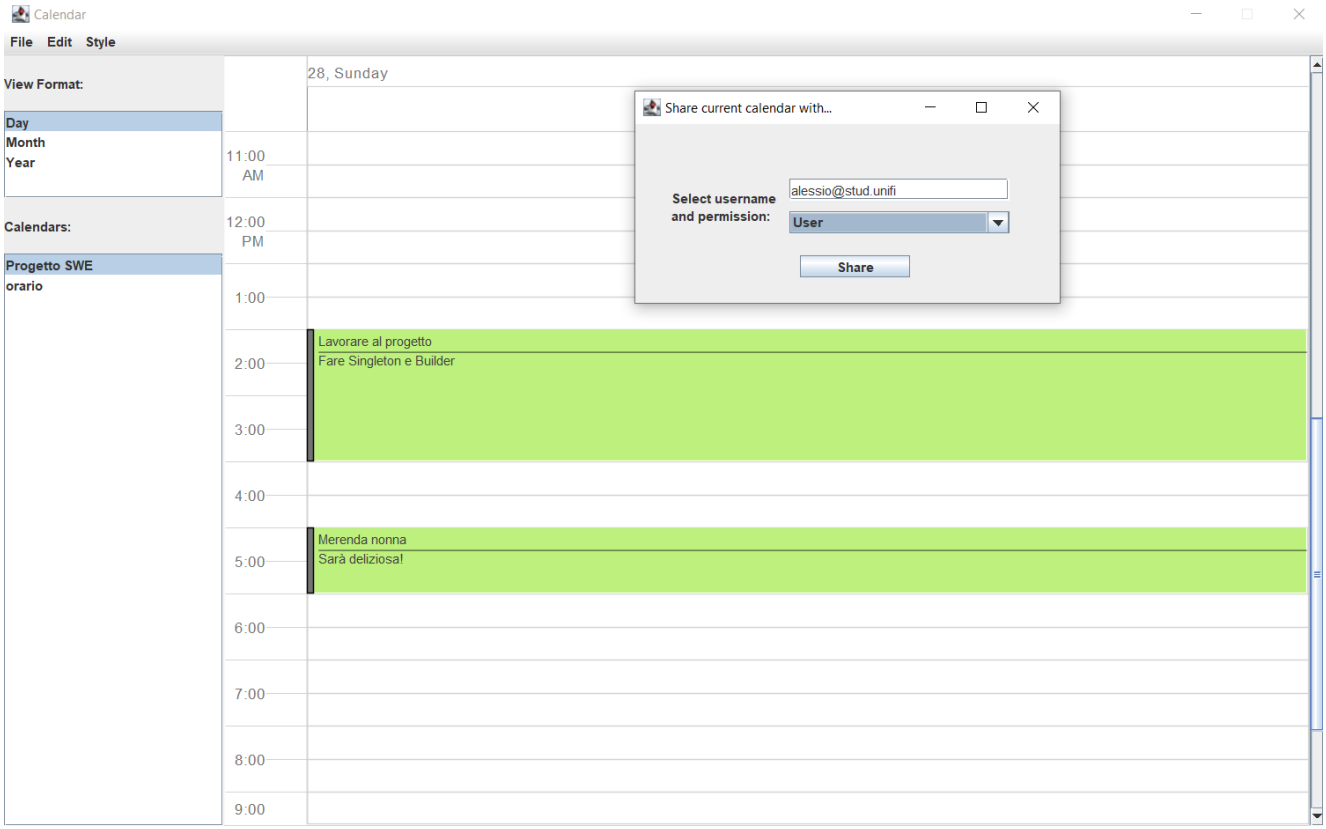

Figura 4.4: Demo - condivisione calendario

Riferimenti

[1] Java Appointment Calendar, Mindfusion,
https://www.mindfusion.eu/java-scheduler.html

[2] UML Class Diagram of Role Based Access Control,
https://www.researchgate.net/figure/UML-model-of-the-Role-Based-Access-Control-Pattern-18_fig5_309261114