UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze

Dipartimento di Ingegneria dell'Informazione

Relazione TownFest

Autori:
Edoardo Nero, Giacomo Casali

N° Matricole:
6301440
6320525

Corso principale:
Ingegneria del Software

Docente corso:
Enrico Vicario

Indice

Elenco delle figure

1 Analisi

In questa sezione andremo ad analizzare il software **TownFest** da noi sviluppato. Partiremo illustrandone gli obiettivi, dando una visione di insieme dei servizi offerti.

1.1 *Statement*

Questo applicativo software ha come obiettivo di consentire la gestione degli spazi pubblici di un'area ben definita per ospitare eventi come fiere o sagre. Sono permesse due tipologie di accesso: la prima come amministratore della piattaforma, riservato ad uno o più impiegati comunali (**IC**), oppure come utenti usufruitori del servizio, i cittadini. Il sistema si avvale di un database per la gestione separata dei dati di ogni utente e di un'interfaccia grafica per usufruire dei servizi offerti. Quest'ultimi sono:

- Aggiunta/modifica/eliminazione dei cittadini registrati
- Aggiunta/modifica/eliminazione di eventi
- Aggiunta/modifica/eliminazione e prenotazione dei padiglioni
- Aggiunta/eliminazione di Licenze per gli utenti
- Visualizzazione dei dati di ogni utente/evento/padiglione
- Prenotazione e verifica dei biglietti associati agli eventi

Gli eventi ed i padiglioni sono visualizzabili graficamente da ogni utente, mentre la loro modifica o aggiunta è permessa solo all'IC. Il cittadino può prenotare un padiglione per un determinato evento se è in possesso di una licenza valida, in caso contrario può inoltrare una richiesta di licenza all'IC.
L'impiegato Comunale è responsabile del rilascio delle licenze ed è in grado di inoltrare una email ai partecipanti di un evento. Ad ogni utente è inoltre permesso richiedere al massimo due biglietti per un determinato evento. La validità di questi può essere controllata dagli IC.

1.2 *Use Case Diagram*

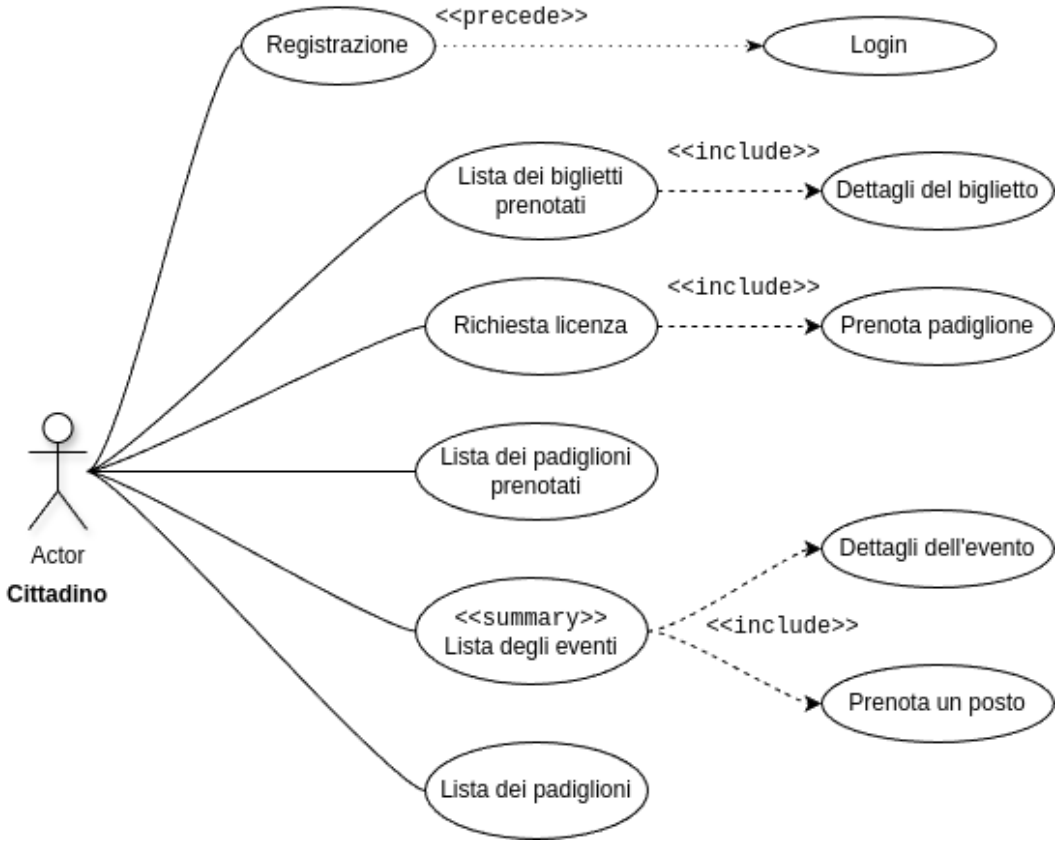

Figura 1: Use Case Diagram Cittadino

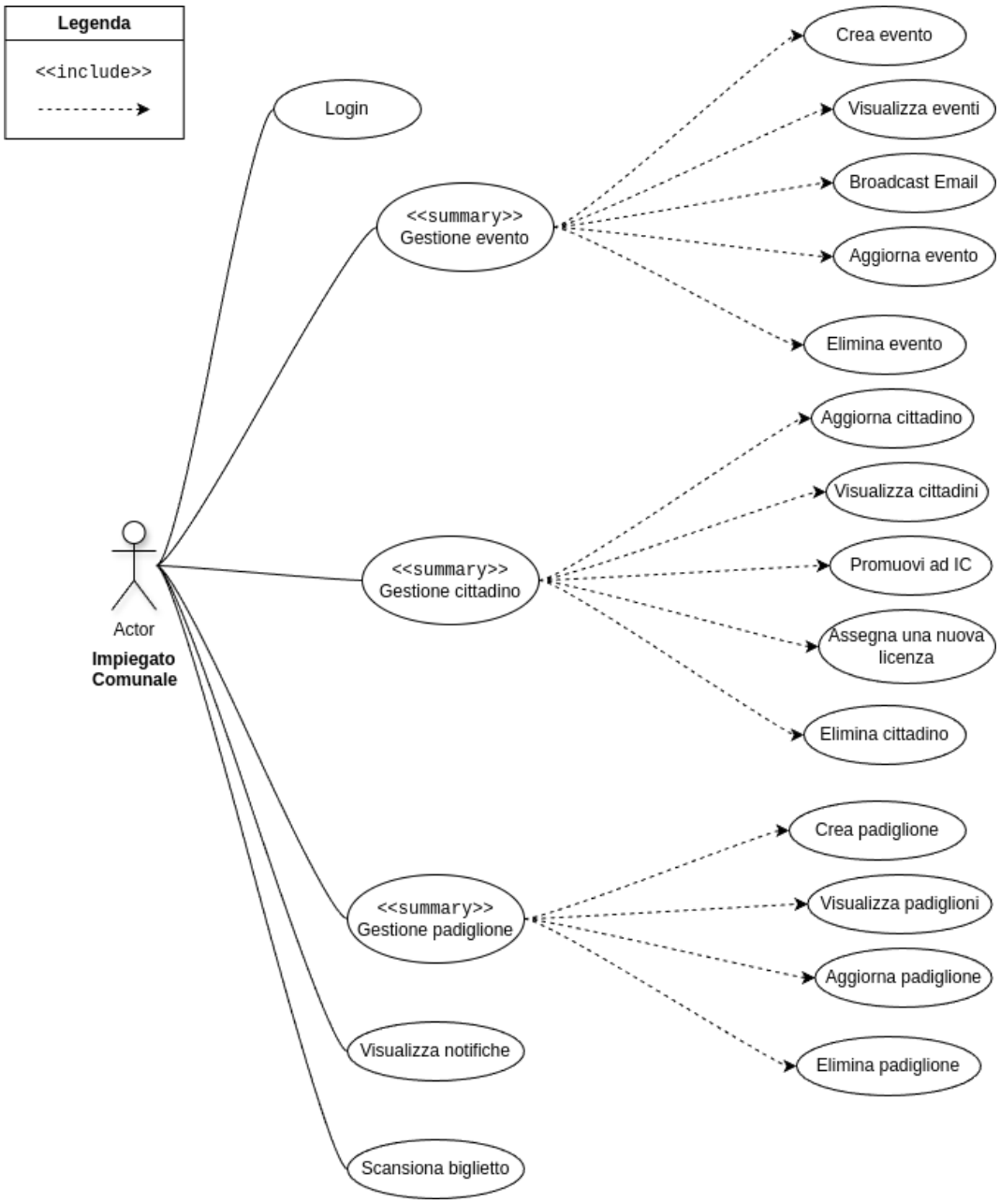

Figura 2: Use Case Diagram Impiegato Comunale

In questi diagrammi lo stereotipo ≪**include**≫ viene qui utilizzato per descrivere la situazione in cui un caso d'uso ne comprende un altro. Nel nostro caso ad esempio quando viene usato nel collegamento tra "Lista Eventi" e "Dettagli Evento" si intende che richiedere la lista degli eventi comprende la richiesta dei dettagli del singolo evento.

Con lo stereotipo ≪**precede**≫ invece andiamo a descrivere il fatto che deve essere terminata l'operazione da cui parte l'associazione prima di poter eseguire l'altra. Ad esempio, in Figura 1, prima di poter eseguire "Login" deve essere stata fatta una "Registrazione".

1.3 *Use Case Template*

Di seguito è riportato un Use Case Template che mostra il flusso di esecuzione di uno casi d'uso rappresentati in Figura 1 e 2. È stato scelto un caso considerato più rappresentativo.

Use Case #1	Prenota Evento
Level	User Goal
Actor	Cittadino
Normal Flow	1. Il cittadino si trova nella pagina principale. 2. Seleziona il menu "Lista eventi". 3. Seleziona un evento dalla lista. 4. Sceglie l'operazione "Prenota". 5. Inserisce i dati personali. 6. Preme conferma. 7. Il software esegue la transazione, la salva nel database e riporta il cittadino alla pagina principale.
Alternative Flow	5_A. Sceglie l'opzione "Annulla" e torna alla lista degli eventi 5_B. L'utente genera dei dati di test. 7_A. Il cittadino ha inserito dati scorretti, il sistema non esegue la transazione e rimane nella pagina di prenotazione

Tabella 1: Prenota Evento

2 Architettura e tecnologie

In questa sezione viene discussa la struttura organizzativa del sistema *software*.

2.1 Interazione tra i moduli

Nel diagramma in Figura 3 sono mostrate graficamente le dipendenze tra i *packages* che contengono il progetto.
L'architettura del programma è organizzata in *business logic*, *domain model* e *ORM*. Quest'ultimo è organizzato in un `DAO` ed `RDBMS`.
Verranno forniti maggiori dettagli nella Sezione 4.

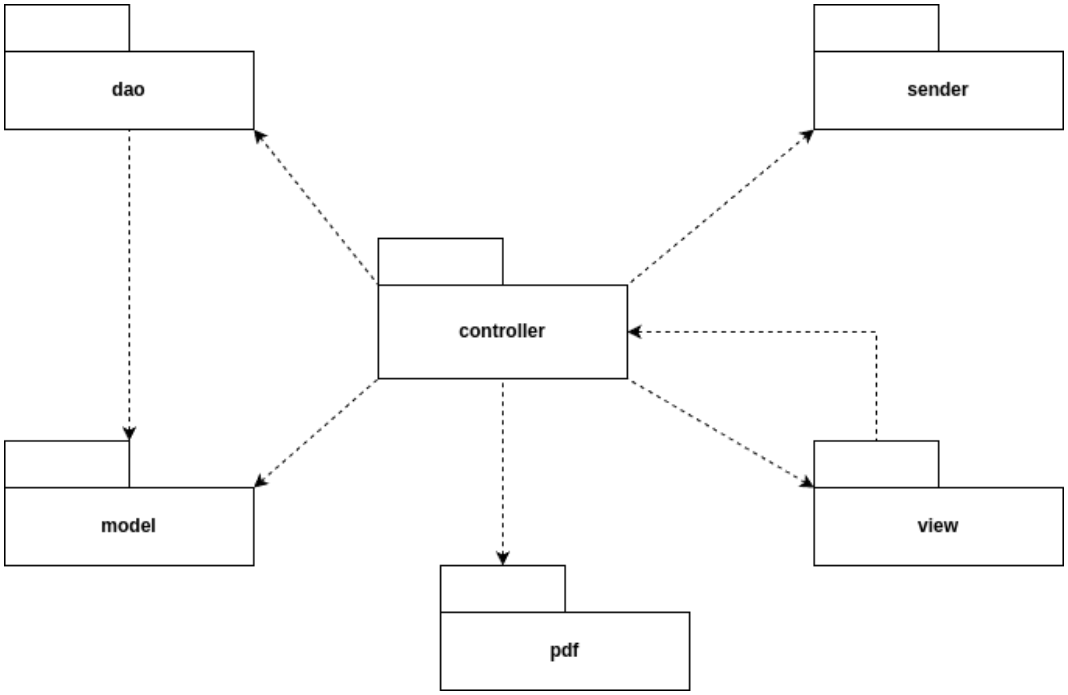

Figura 3: Schema interazione tra i moduli

2.2 Interfaccia utente

L'utente interagisce con il programma attraverso una *Graphic User Interface* (GUI). In questo modo egli può navigare tra i vari menu presenti e scegliere le operazioni che desidera eseguire.
Il programma offre visualizzazioni personalizzate per ogni utente, tenendo anche in considerazione se si tratta di un amministratore o meno (IC). Questo tipo di interazione è resa possibile grazie alla libreria JavaFX.

2.3 Memorizzazione dei dati

In questa sezione viene descritta l'architettura del *database* di cui fa uso il programma e le tecnologie utilizzate per realizzarlo.

2.3.1 Schema *Entity Relationship*

Per memorizzare e gestire in modo opportuno i dati di ciascun utente il programma utilizza un *database*, organizzato nelle seguenti tabelle:

- `utente`
- `evento`
- `licenza`

- `padiglione`
- `biglietto`
- `notifica`
- `padiglione_evento`

Le relazioni tra le entità e i relativi vincoli sono rappresentati in Figura 4 utilizzando il modello *Entity Relationship*.

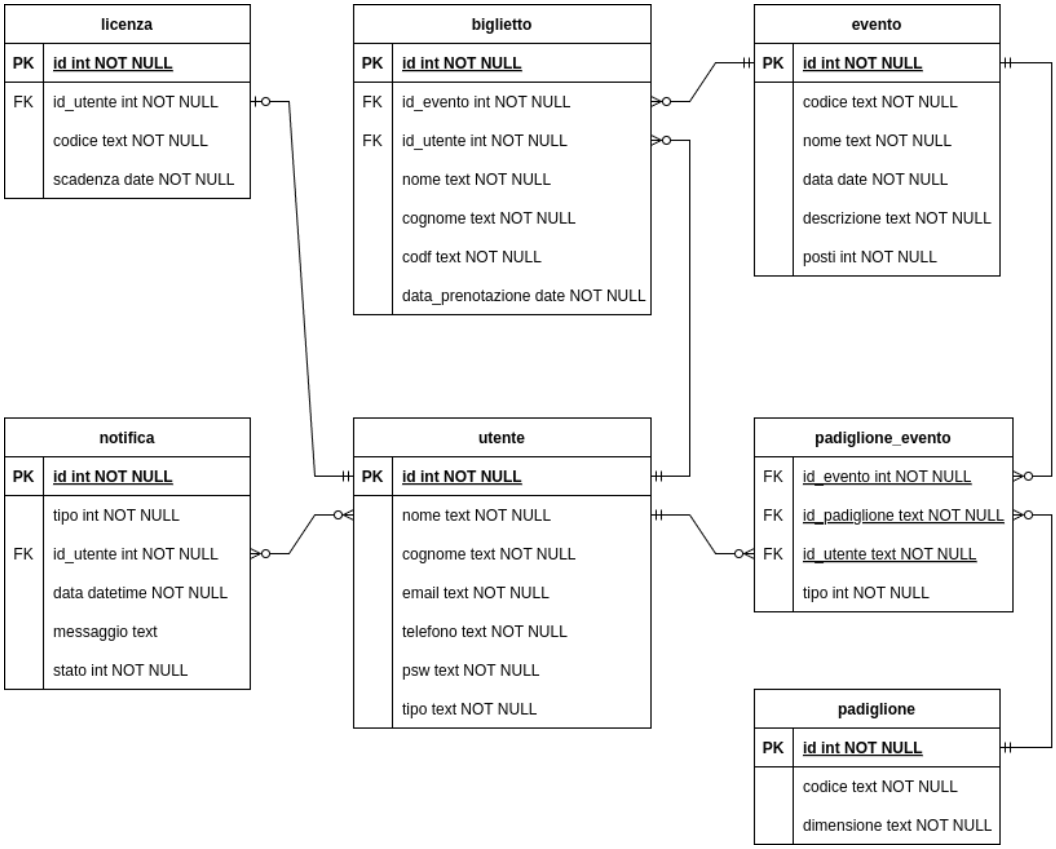

Figura 4: Modello Entity Relationship del database implementato

2.3.2 Schema logico e tecnologie utilizzate

Il *database* è stato creato con nome `town_fest` attraverso PHPMyAdmin e utilizzando un server Apache locale come *host*. La gestione è stata fatta tramite strumenti integrati nell'editor *IntelliJ IDEA Ultimate* e tramite l'interfaccia a riga di comando utilizzando il comando `mysql`.
La comunicazione con il *database* all'interno del programma è effettuata utilizzando le librerie JDBC. Esse sono dotate di API che consentono di connettersi al *database* ed eseguire query tramite il linguaggio Java.

2.4 Tecnologie extra e librerie esterne

Il nostro sistema, come accennato in precedenza, permette all'utente di prenotare biglietti per i vari eventi presenti.
Ad ogni utente è permesso salvare i propri biglietti come file in formato PDF tramite la libreria java `PDFJet`. Questi file contengono un codice a barre identificativo generato tramite la libreria `Barbecue` e diverso per ogni biglietto.
Per rendere dunque il progetto ipoteticamente funzionante abbiamo anche realizzato un piccolo applicativo web in PHP e JavaScript utilizzabile da tutti gli IC addetti al controllo dei biglietti all'ingresso degli eventi. Questo strumento è di supporto al software principale e permette tramite un comune smartphone di verificare il biglietto, altrimenti verificabile tramite un PC caricando il file PDF.

Per controllare la validità tramite smartphone abbiamo usato una libreria di nome `Quagga`, la quale permette la lettura di codici a barre tramite la fotocamera del dispositivo.

2.4.1 Codice Sorgente

Tutto il codice da noi scritto ed i file relativi a questo progetto sono presenti sulla piattaforma GitHub (https://github.com/GCasaliUnifi/ProgettoSWE_Casali_Nero). Il codice è visibile a tutti ed è modificabile sotto licenza MIT.

3 Progettazione

3.1 *Page Navigation Diagram*

In questo paragrafo vengono mostrati due diagrammi (Figure 5, 6) che rappresentano la logica di navigazione tra le pagine del sistema, una per l'IC e una per il cittadino.
Le pagine indicate nelle immagini come `Dashboard`, `LogIn`, `Prenota Evento` e `Verifica Biglietto` sono mostrate nelle immagini della Sezione 3.2

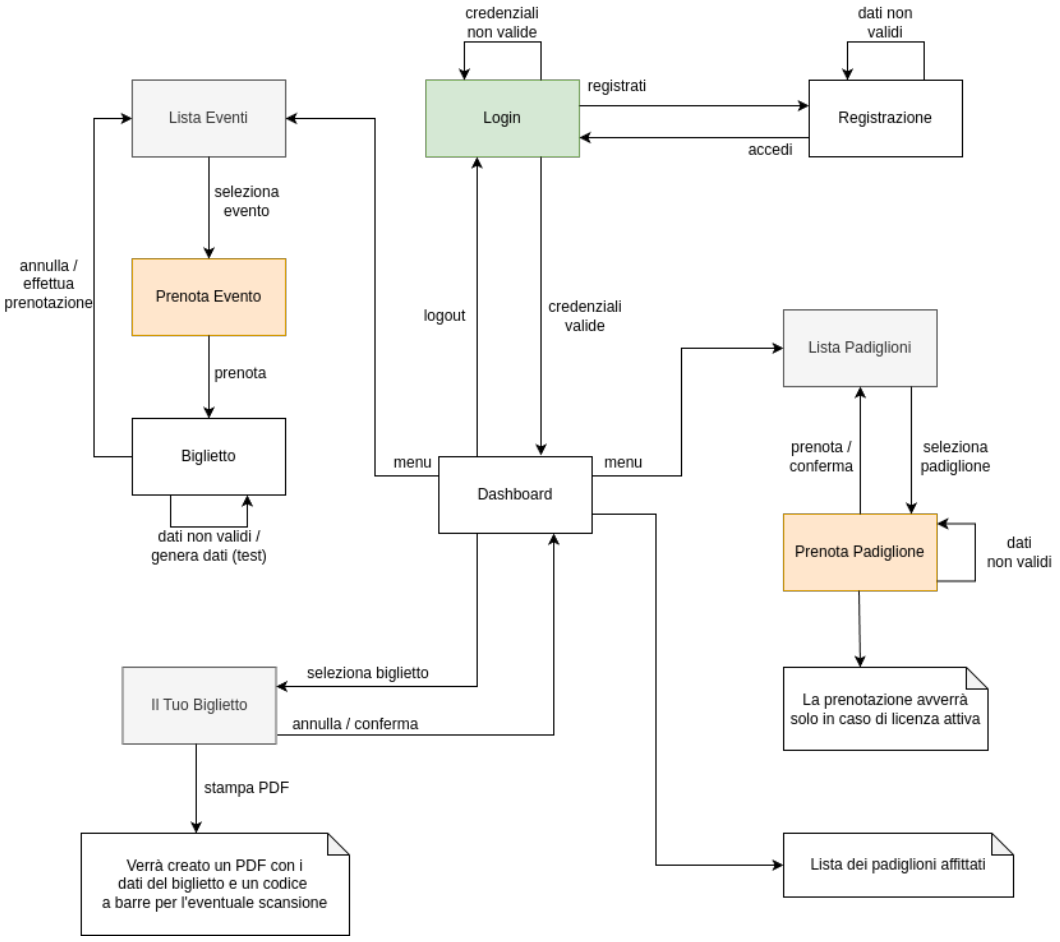

Figura 5: Page Navigation Diagram del cittadino

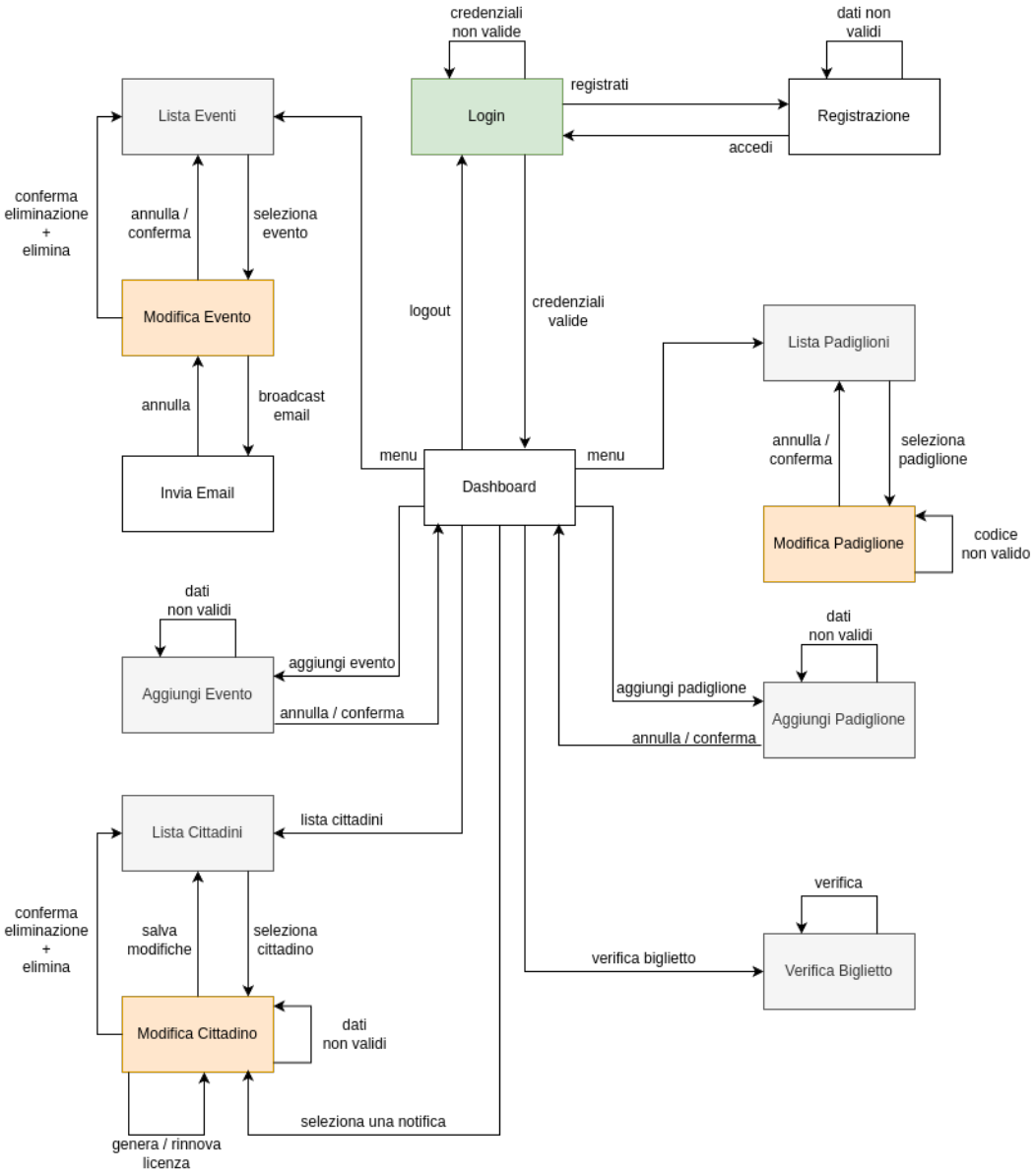

Figura 6: Page Navigation Diagram dell'IC

3.2 *Interfaccia grafica*

In questa Sezione vengono mostrate alcune schermate della GUI da noi implementata, come citato in 2.2

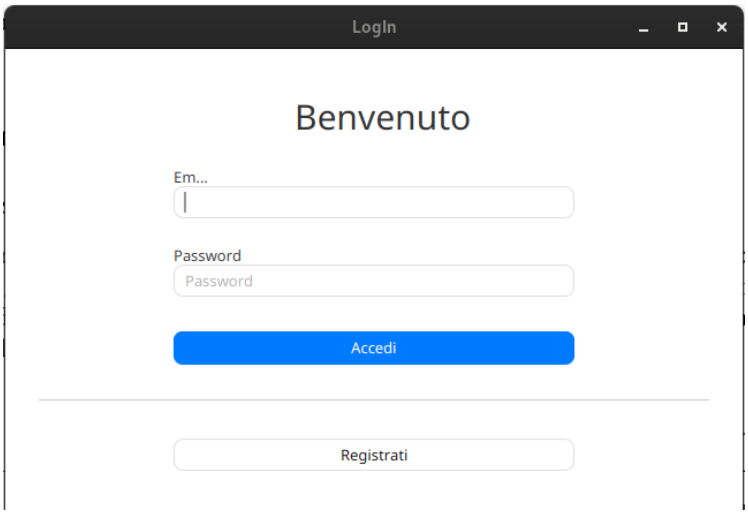

Figura 7: *Schermata di LogIn*

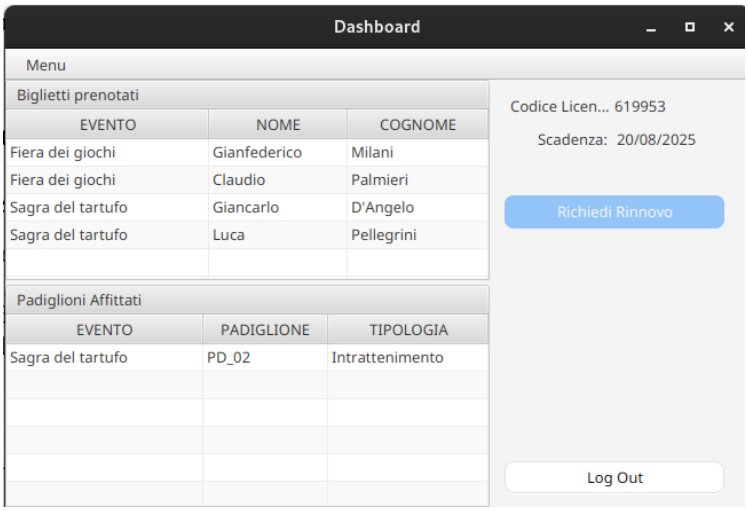

Figura 8: *Schermata della Home*

Figura 9: *Schermata prenotazione Evento*

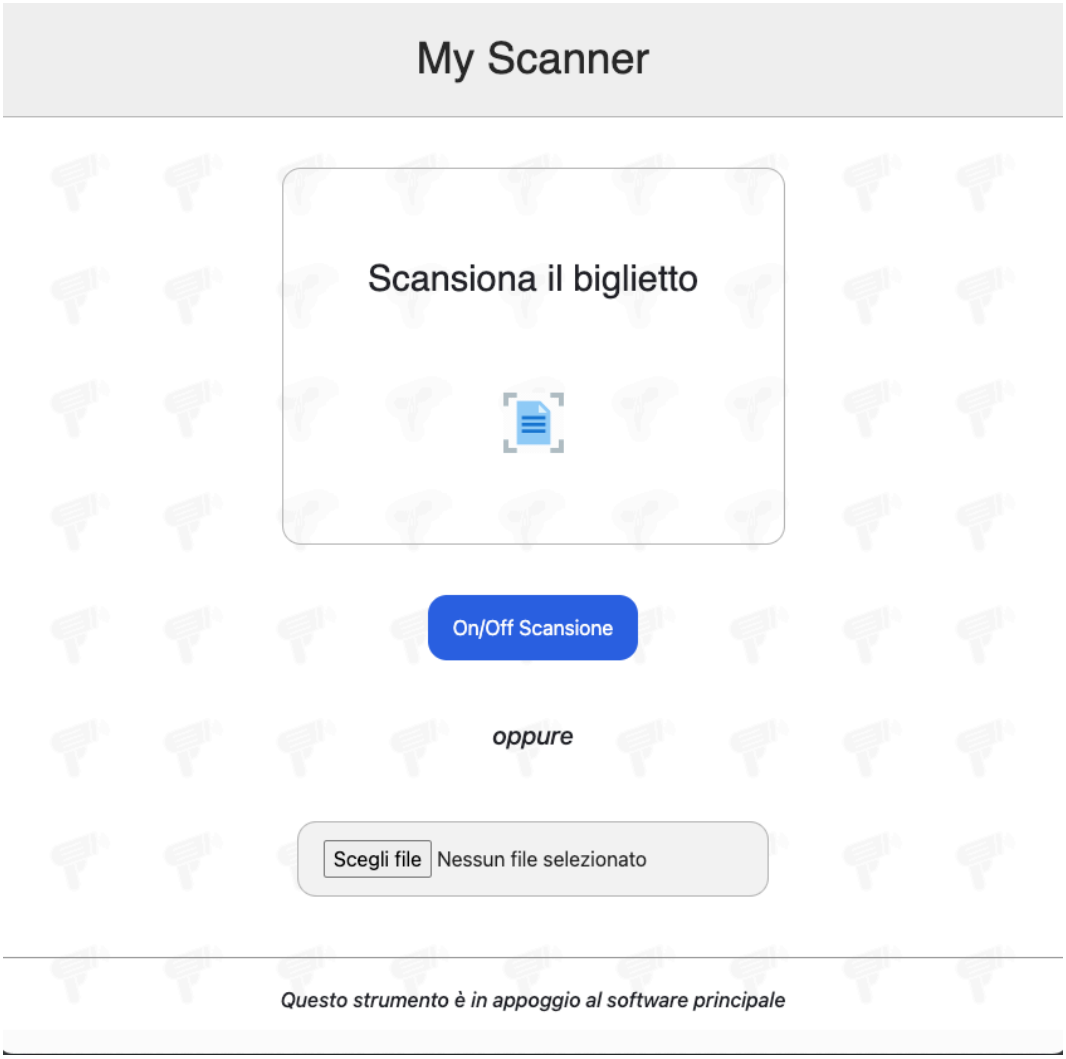

Figura 10: *Schermata da smartphone della Verifica Biglietto*

3.3 *Class Diagram*

In Figura 11 è rappresentato il diagramma delle classi relativo all'intero progetto. Le classi sono suddivise in diversi *package* in base alla tipologia del servizio offerto. Per maggiore chiarezza in questo diagramma vengono omessi i metodi e gli attributi che compongono le classi, eccezion fatta per quelli più significativi ai fini della comprensione del funzionamento del programma. Le classi che compongono ciascun *package* verranno discusse nel dettaglio nella Sezione 4.2, mentre i dettagli sulle dipendenze sono affrontati alla Sezione 4.1.

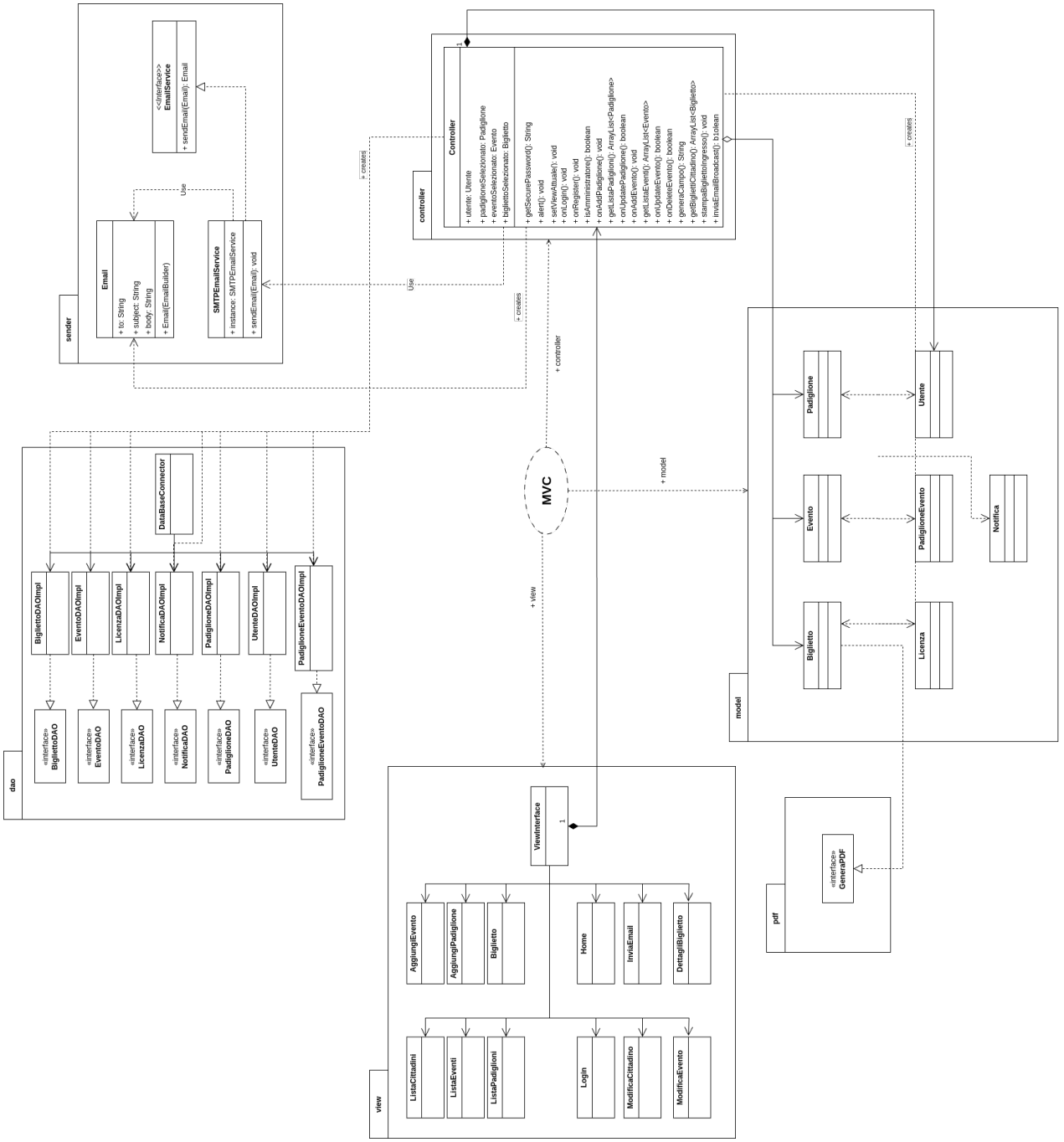

Figura 11: *Class Diagram*

4 Implementazione

In questa sezione discuteremo i dettagli implementativi del nostro progetto, concentrandoci sulle caratteristiche più interessanti e spiegando le motivazioni che ci hanno spinto ad adottare determinate soluzioni.
L'intero programma è stato realizzato in Java, eccezion fatta della componente web descritta nella Sezione 2.4

4.1 *Packages*

In questa sezione vengono affrontate nel dettaglio le caratteristiche di implementazione principali dei *packages* utilizzati e le loro dipendenze. Nel nostro progetto sono presenti *packages* "Personalizzati", ovvero implementati da noi, altri facenti parte della Java Standard Library che sono forniti insieme al linguaggio Java ed infine sono presenti anche *packages* fortniti dalle librerie esterne da noi usate per alcune delle funzioni del nostro applicativo.

4.1.1 *Packages* "personalizzati"

I *packages* implementati da noi sono i seguenti:

- `view`
- `model`
- `controller`
- `dao`
- `sender`
- `pdf`

Il *package* `view` contiene tutte le classi che permottono l'interfaccia tra l'utente e il software. Ogni classe al suo interno fa uso della lireria JavaFX per poter rendere visibili a schermo delle GUI che consentono agli utenti di scegliere le operazioni da eseguire.

Il *package* `model` rappresenta il *domain model* del programma, ovvero le classi che rappresentano le entità del dominio applicativo di questa applicazione.

Il *package* `controller` contiene tutta la *business logic* del programma. I suoi compiti principali sono quelli di gestire le richieste dell'utente, eventualmente connettendosi al *database*, e quello di aggiornare l'interfaccia garantendo che vengano mostrate le informazioni corrette all'utente.

Il *package* `dao` contiene le classi che permettono al programma di interfacciarsi al *database*. Queste vengono istanziate all'interno dei metodi della classe `Controller` e vengono utilizzate solo all'interno del metodo per poi essere distrutte. Questa scelta è stata presa per ridurre al minimo il tempo di connessione al *database*.

Il *package* `sender` contiene quelle classi che permettono l'invio di email. Le sue classi vengono usate solo all'interno dei metodi del controller.

Il *package* `pdf` contiene una sola classe che ha come compito quello di generare su richiesta un file PDF rappresentativo di un determinato biglietto.

Le classi che compongono ciascun *package* verranno trattate nel dettaglio successivamente nella Sezione 4.2.

4.1.2 *Java Standard Libraries* e librerie esterne

Nel programma vengono utilizzati diversi *packages* provenienti delle librerie standard di Java e da alcune librerie esterne non incluse tra quelle standard. Quelli più importanti sono:

- `java.time`: usato dal `controller` per gestire la *busines logic* delle date.

- `Java DataBase Connectivity (JDBC)`: viene utilizzato nel *package* `dao` per gestire la connessione al *database* e le operazioni che lo coinvolgono (CRUD).
- `JUnit`: viene utilizzato per la parte di *Unit Testing* del *software* (vedere Sezione 5)
- `JavaFX` : questa viene utilizzata dal *package* `view` per creare la GUI con cui interagisce l'utente.
- `PDFjet` : utilizzata per permettere al cittadino di scaricare i propri biglietti come un file PDF
- `barbecue-1.5` : questa libreria viene utilizzata per effettuare la scansione del codice a barre presente in ogni biglietto tramite la fotocamera del dispositivo che si collega all'applicativo web citato nella Sezione 2.4.
- `Quagga`: usata per permettere la lettura dei codici a barre tramite fotocamera.

4.2 Classi

In questa sezione vengono descritte le classi implementate suddivise in base al *package* a cui appartengono.

4.2.1 `Model`

Le classi contenute nel *package* `model` racchiudono le strutture dati che rappresentano le varie entità del *domain model* del programma. Segue dunque un elenco delle classi e delle loro principali caratteristiche:

- `Utente`: classe che contiene i dati dell'utente che rappresenta.
- `Padiglione`: contiene i dati relativi ad un padiglione.
- `Evento`: contiene informazioni di un evento tra cui data, descrizione e nome.
- `Licenza`: classe con i dati di una licenza.
- `Biglietto`: classe contenente tutti i dati relativi ad un biglietto.
- `Notifica`: classe con le informazioni delle notifiche.
- `PadiglioneEvento`: classe che rappresenta la tabella del *database* che ci dice quali padiglioni sono prenotati per un evento.

Ognuna di queste classi espone metodi per assegnare e reperire singolarmente ogni suo attributo. Dato che abbiamo implementato il *pattern DAO*, queste classi non presentano nessun codice di collegamento con il *database*.

4.2.2 *View*

Qui sono presenti tutte quelle classi che utilizzano le librerie di `JavaFX` per mostrare i dati e le operazioni eseguibili attraverso una GUI. Qui è presente una classe astratta `ViewInterface` che viene derivata da tutte le altre classi. Essa definisce il metodo astratto `display()` che ogni classe derivata deve implementare, poiché necessario per visualizzare ogni finestra in JavaFX. Sempre perché richiesto da JavaFX, ogni classe deve tenere un riferimento ad uno "`Stage`" e alla finestra padre `Parent`, dunque li abbiamo inseriti in questa classe in modo da poterli utilizzare in ogni classe derivata.
Segue qua la lista di ogni classe presente dentro al *package* `View`. I nomi delle classi rispecchiano la funzione principale esposta all'utente dalla finestra corrispondente, dunque viene omessa una descrizione del compito della classe quando non necessaria:

- `Home`: Finestra principale esposta all'utente dopo il LogIn. Questa presenta vari pulsanti e menu che permettono di selezionare le varie operazioni disponibili all'utente corrente. Presenta oltre una serie di informazioni relative all'utente connesso, come la scadenza della licenza o la lista dei biglietti acquisiti.
- `Biglietto`: Finestra che, quando viene richiesto, viene usata per prenotare un biglietto. Per scopi di test è presente anche un pulsante che genera i dati che riempono i campi del biglietto in automatico

- `LogIn` e `Registrazione`
- `AggiungiEvento`
- `AggiungiPadiglione`
- `DettagliBiglietto`
- `InviaEmail`
- `ListaCittadini`
- `ListaEventi`
- `ListaPadiglioni`
- `ModificaCittadino`
- `ModificaPadiglione`

JavaFX fa riferimento a dei file di tipo `fxml` per posizionare i vari elementi nella GUI. Per comprenderne meglio il funzionamento, nelle figure successive vengono mostrate una parte della classe `LogIn` e l'implementazione del metodo astratto `display`.

```
public abstract void display() throws IOException;
```

Figura 12: *Metodo astratto di ViewInterface*

```
public class LogIn extends ViewInterface {

    @FXML
    private TextField email;
    @FXML
    private TextField psw;
    @FXML
    private Button accedi;
    @FXML
    private Button registrati;

@FXML
public void initialize() {
    accedi.setOnAction(event -> {
        if(email.getText().isEmpty() || psw.getText().isEmpty()) {
            System.out.println("Errore: uno o più campi sono vuoti!");
        } else {
            try {
                controller.onLogin(email.getText(), psw.getText());
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }
    });

    registrati.setOnAction(event -> {
        try {
            controller.setViewAttuale(new Registrazione(this.controller, stage));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    });

@Override
public void display() throws IOException{
    try {
        FXMLLoader loader = new FXMLLoader(getClass().getResource(fxmlPath));
        loader.setController(this);
        Parent root = loader.load();
        Scene scene = new Scene(root, width, height);
        stage.setScene(scene);
        stage.setTitle(title);
        stage.show();
    } catch (IOException e) {
        throw new IOException("Errore nel caricamento della finestra di login");
    }
}
```

Figura 13: *Metodi principali Classe LogIn*

4.2.3 *Controller*

Il *package* `controller` contiene solo l'omonima classe `Controller`.
La classe ha un ruolo cruciale per il corretto funzionamento del programma e svolge le seguenti funzioni:

- Gestione delle operazioni relative alle scelte effettuate dall'utente attraverso i menu.

- Interfaccia con il *database* per tutte le operazioni che necessitano l'aggiunta, la modifica, rimozione o lettura di dati da esso.
- Operazioni di *business logic* sui dati ogni volta che viene effettuata una modifica al *database*.
- Aggiornamento della finestra GUI attuale e dei dati al suo interno.

Tutta la *business logic* del programma risiede in questa classe. Essa mantiene un riferimento ad un oggetto di tipo `ViewInterface`, che rappresenta la GUI attualmente esposta all'utente, e mantiene anche un riferimento ad un `Utente` che contiene i dati dell'utente che ha effettuato il LogIn.
Questa classe inoltre fa uso delle classi del *package* `dao`, istanziandole temporanemente all'interno dei metodi che lo richiedono in modo da mantenere la connessione al *database* soltanto per il tempo necessario ad eseguire le operazioni richieste.

4.2.4 *DAO*

Ciascuna classe DAO all'interno di questo package ha il compito di fare da intermediario tra il software e il database. Abbiamo strutturato il *package* con una classe DAO per ogni struttura dati. Ciascuna di queste classi implementa le operazioni di base per la gestione dei dati (metodi **CRUD**). I metodi corrispondono alle operazioni `SQL` di `INSERT`, `UPDATE`, `DELETE` e `SELECT`.
Il package contiene le classi:

- `DataBaseConnector`: è la classe utilizzata da tutte le altre presenti nel package, tramite `extends`, per stabilire una connessione con il RDBMS.
- `BigliettoDAO`: è la classe con la quale il software si interfaccia con la tabella "biglietto".
- `EventoDAO`: è la classe con la quale il software si interfaccia con la tabella "evento".
- `LicenzaDAO`: è la classe con la quale il software si interfaccia con la tabella "licenza".
- `NotificaDAO`: è la classe con la quale il software si interfaccia con la tabella "notifica".
- `PadiglioneDAO`: è la classe con la quale il software si interfaccia con la tabella "padiglione".
- `PadiglioneEventoDAO`: è la classe con la quale il software si interfaccia con la tabella "padiglione_evento".
- `UtenteDAO`: è la classe con la quale il software si interfaccia con la tabella "utente".

La connessione con il *database* che viene stabilita da ciascun metodo di queste classi viene chiusa prima di terminare la chiamata. In questo modo si evita il rischio di effettuare troppe connessioni contemporaneamente e si mantiene la connessione solo per il tempo strettamente necessario a completare l'operazione.

4.2.5 *Sender*

Questo *package* contiene delle classi che vengono usate dall'IC per poter mandare delle mail ad ogni utente che ha un bigliettp per un determinato evento. Al suo interno sono presenti le seguenti classi:

- `Email`: Rappresenta la generica Email e possiede i campi tipici della posta elettronica.
- `EmailBuilder`: Classe che tramite il pattern *Builder* ci permette di generare elementi di tipo `Email` passo dopo passo in modo flessibile.
- `EmailService`: classe di interfaccia da cui devono derivare tutti servizi resi disponibili (SMTP, POP3, etc...).
- `SMTPEmailService`: classe che rappresenta l'unico servizio da noi reso disponibile di invio mail. In questa classe si fa uso del *design pattern* `Singleton`.

4.3 Dettagli progettazione

Dopo aver discusso le classi implementate spieghiamo adesso i dettagli di progettazione, con particolare attenzione ai *Design Pattern* utilizzati, che sono:

- `Model View Controller`
- `Data Access Object (DAO)`
- `Builder`
- `Singleton`

4.3.1 *Model View Controller*

Il *Model View Controller*, abbreviato MVC, è un *design pattern* architetturale che permette la separazione delle responsabilità e una struttura modulare. Esso suddivide l'applicazione in tre parti principali: il *Model*, la *View* e il *Controller*. Abbiamo affidato a ciascuna parte le seguenti responsabilità:

- *Model*: gestisce l'accesso, la manipolazione e la rappresentazione dei dati. Il Model è responsabile di rappreentare e conservare lo stato dell'applicazione. Il Model notifica al Controller eventuali cambiamenti di stato.
- *View*: gestisce la presentazione delle informazioni dell'interfaccia utente. Riceve l'input dall'utente e, comunicando con il Controller, riceve dati dal Model e visualizza i risultati. Questa componente non contiene né gestisce la *business logic* del sistema.
- *Controller*: funge da intermediario tra il *Model* e la *View*. Gestisce l'input dell'utente e decide come interagire con il Model, inviando anche aggiornamenti alla View.

Nel nostro progetto il modello e la vista sono naturalmente separati. Infatti l'utente comunica con il *software* attraverso la GUI, mentre i dati su cui deve operare si trovano su di un *database*. Di conseguenza ci è sembrato utile ed efficace utilizzare un'entità frapposta tra queste componenti, il *Controller*, che si occupi di gestire la comunicazione tra utente e *database*, garantendo che i dati siano sempre completi e aggiornati.
Il flusso di esecuzione avviene in questo modo:

1. L'utente naviga tra i menu della GUI e sceglie l'operazione da eseguire. A questo punto viene quindi chiamato un metodo corrispondente del *package* `view` che gestisce l'operazione.
2. Il metodo precedente richiama il `Controller` che crea una rappresetnazione del `Model` interfacciandosi con il *database* attraverso le classi del *package* `dao` e mappando i dati in istanze delle classi del *package* `model`.
3. Il `Controller` esegue dunque l'operazione richiesta e aggiorna, se necessario, il modello sul *database*.
4. Il `Controller` aggiorna dunque la GUI per mostrare i dati aggiornati o la finestra corretta se l'operazione prevede un cambio di scena.

Sulla base di queste considerazioni abbiamo scelto di associare alla *View* tutto ciò che mostra un'interfacia all'utente, ovvero le classi del *package* `view`. Abbiamo invece associato al *Model* tutte le classi facenti parte del *package* `model`.

4.3.2 *Data Access Object*

Il *Data Access Object* è un *pattern* architetturale utilizzato per separare la *business logic* dalla logica di accesso ai dati. Questo pattern consente di isolare le operazioni di lettura e scrittura verso il *database*, mantenendo l'interazione incapsulata all'interno di oggetti dedicati. In questo modo la *business logic* non dipende direttamente dall'implementazione del database, rendendo il codice più facile da mantenere e testare.

4.3.3 *Builder*

Il pattern `Builder` è un pattern creazionale che separa la costruzione di un oggetto complesso dalla sua rappresentazione, consentendo di creare oggetti passo dopo passo con maggiore flessibilità. Viene usato nel *package* `sender` per gestire la creazione delle email da inviare.

4.3.4 *Singleton*

Il pattern `Singleton` è un pattern creazionale che garantisce che una classe abbia una sola istanza globale e fornisce un punto di accessi centralizzato a quell'istanza, evitando la creazione multipla dello stesso oggetto. Dato che in futuro vorremmo dotare il sistema di molteplici servizi di invio Email, ci è sembrato corretto utilizzare questo pattern per garantire che solo uno di essi sia istanziato dal programma per inviare le mail.

5 *Unit Testing*

Il *testing* del programma è stato effettuato tramite la libreria `JUnit`. Per questo progetto sono state testate solo le funzionalità ritenute più critiche per il corretto funzionamento del programma. Tramite le direttive di JUnit abbiamo testato le funzionalità principali della classe `Controller` delle classi presenti nei *package* `model` e `sender`.
A titolo di esempio, si mostra nella figura 14 uno dei test usati per il `sender`, di nome `SMTPEmailServiceTest.java`:

```java
package test.sender;

import org.junit.Test;
import sender.Email;
import sender.SMTPEmailService;

import static org.junit.Assert.*;

public class SMTPEmailServiceTest {

    @Test
    public void testSingletonInstance() {
        SMTPEmailService instance1 = SMTPEmailService.getInstance();
        SMTPEmailService instance2 = SMTPEmailService.getInstance();
        assertSame(instance1, instance2);
    }

    @Test
    public void testSendEmail() {
        SMTPEmailService emailService = SMTPEmailService.getInstance();
        Email email = new Email.EmailBuilder()
                .setTo("example@example.com")
                .setSubject("Subject")
                .setBody("Email body")
                .build();

        emailService.sendEmail(email);
        // Verifica che l'email sia stata inviata correttamente (puoi usare un mock per questo)
```

Figura 14: *codice test per STMPEmailService*