­

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Object Design Document  Progetto  SalernArte   |  |  | | --- | --- | | Versione | 0.1 | | Data | 28/05/2022 | | Destinatario | Studenti di Ingegneria del Software 2021/22 | | Presentato da | Martino Lucia [0512105234], Longo Marco [0512105945], Della Pepa Alessia [0512105720] | | Approvato da |  | |

Membri del Team

|  |  |  |  |
| --- | --- | --- | --- |
| **Nome** | **Matricola** | **Acronimo** | **Informazioni di contatto** |
| Della Pepa Alessia | 0512105720 | DPA | a.dellapepa5@studenti.unisa.it |
| Longo Marco | 0512105945 | LM | m.longo36@studenti.unisa.it |
| Martino Lucia | 0512105234 | ML | l.martino11@studenti.unisa.it |

RevisionHistory

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Descrizione** | **Autori** |
| 28/05/2022 | 0.1 | Prima stesura: struttura del documento e inserimento del package SalernArte | DPA, ML, LM |
| 30/05/2022 | 0.2 | Inserimento package gestione acquisti e class interface package gestione acquisti | LM |
| 06/06/2022 | 0.3 | Inserimento introduzione, object design trade-off, linee guida per la documentazione dell’interfaccia, introduzione alla sezione packages | DPA, ML, LM |
| 07/06/2022 | 0.4 | Inserimento package  Registrazione e Autenticazione.  Inserimento class interface  Package Registrazione e  Autenticazione | ML |
| 09/06/2022 | 0.5 | Aggiunta Class Diagram UtenteRegistrato/Scolaresca, package control e model | ML |
| 10/06/2022 | 0.6 | Inserimento package gestione eventi, class interfaces per la gestione eventi e class diagram per gestione eventi | DPA |
| 11/06/2022 | 0.7 | Inserimento class diagram amministratore e organizzatore, package control GestioneAcquisti | LM |
| 13/06/2022 | 0.8 | Inserimento del glossario | LM |

Sommario

[Membri del Team 2](#_Toc104628762)

[RevisionHistory 2](#_Toc104628763)

[1. Introduzione 4](#_Toc104628764)

[1.1. Object Design goals 4](#_Toc104628765)

[1.2. Linee guida per la documentazione dell’interfaccia 4](#_Toc104628766)

[1.3. Definizioni, Acronimi e Abbreviazioni 4](#_Toc104628767)

[1.4. Riferimenti 4](#_Toc104628768)

[2. Packages 4](#_Toc104628769)

[3. Class Interfaces 6](#_Toc104628770)

[7. Design Patterns 6](#_Toc104628771)

[8. Glossario 6](#_Toc104628772)

1. Introduzione

SalernArte si propone come obiettivo principale di realizzare un’agenzia online specializzata nella vendita di biglietti riguardanti mostre d’arte ed eventi teatrali e culturali nel salernitano. L’obiettivo è quello di facilitare a tutti i cittadini la ricerca di iniziative culturali salernitane, raggruppando queste ultime in un unico ambiente semplice ed intuitivo, e di ottimizzare il lavoro di organizzatori di eventi che si interfacciano ai cittadini.

In questa prima sezione del documento, verranno descritti i trade-offs e le linee guida per la fase di implementazione, riguardanti la nomenclatura, la documentazione e le convenzioni sui formati.

## Object Design Trade-off

**Comprensibilità vs Tempo:**

Il codice del sistema deve essere comprensibile, in modo da facilitare la fase di testing ed eventuali future modifiche da apportare. Al fine di rispettare queste linee guida il codice sarà integrato da commenti volti a migliorarne la leggibilità; tuttavia, questo richiederà una maggiore quantità di tempo necessario per lo sviluppo del nostro progetto.

**Sicurezza vs Costi:**

La sicurezza rappresenta uno degli aspetti principali del sistema. Tuttavia, a causa di tempi di sviluppo molto limitati, ci limiteremo ad implementare sistemi di sicurezza basati su e-mail e password.

**Sviluppo rapido vs Features:**

Le funzionalità specifiche dell’applicazione verranno realizzate seguendo un sistema basato su delle priorità. Privilegiando uno sviluppo rapido, verrò data la precedenza agli elementi che dispongono di una priorità alta per poi integrare le restanti funzionalità in un secondo momento.

**Riusabilità vs Efficienza**

Il sistema deve essere costruito per garantirne la riusabilità, attraverso l’utilizzo di design pattern, e componenti COTS nonostante ciò potrebbe portare ad un aumento dei tempi di risposta.

## Linee guida per la documentazione dell’interfaccia

Le linee guida includono una lista di regole che gli sviluppatori dovrebbero rispettare durante la progettazione delle interfacce.

### 1.2.1 Nomi dei file

* Le classi devono avere nomi al singolare
* I nomi delle classi devono cominciare con la lettera maiuscola
* I nomi del file sorgente Java devono coincidere con il nome della classe
* I nomi dei file sorgenti Java e in generale dei file devono rappresentare il contenuto del file

### Strutture dei file sorgente

Il progetto dovrà essere strutturato nel seguente modo:

* **.idea**
* **src** , contiene tutti i file sorgente
  + **main**
    - **java,** contiene le classi java relative alle componenti Control e Model
    - **resources,** contiene i file relativi alle componenti View
      * **static,** contiene i fogli di stile CSS e gli script JS
      * **templates,** contiene i file HTML
  + **test,** contiene tutto il necessario per il testing
    - **java,** contiene le classi java per l’implementazione del testing

Nello specifico dovranno rispettare le seguenti strutture:

* I file relativi all’implementazione del sistema seguono la seguente struttura: *src/main/java/subsystem/package/file.java*
* I file relativi alla view seguono la seguente struttura: *src/main/resources/templates/file.html*
* I file di stile CSS e JS seguono la seguente struttura: *src/main/resources/static/file*
* I file relativi al testing seguono la seguente struttura: *src/test/java/subsystem/package/fileTest.java*

### Nomenclatura

* **Package:** lowerCamelCase, solo lettere (MODIFICARE ANCHE TUTTI GRAFI DI QUESTO DOCUMENTO)
* **Classi:** UpperCamelCase, solo lettere
* **Metodi:** lowerCamelCase, solo lettere
* **Costanti:** COSTANT\_CASE, solo lettere e underscore
* **Variabili:** lowerCamelCase, solo lettere
* **Parametri di metodi:** lowerCamelCase

### 1.2.4. Convenzioni metodi DAO

* I metodi del DAO per salvare istanze nel database seguiranno la nomenclatura:

doSave**NomeDellaClasse**

* I metodi del DAO per prelevare istanze nel database basandosi sulla chiave seguiranno la nomenclatura: doRetrieve**NomeDellaClasse**
* I metodi del DAO per prelevare istanze nel database basandosi su un parametro diverso dalla chiave seguiranno la nomenclatura: doRetrieve**NomeDellaClasse**By**Parametro**
* I metodi del DAO per cancellare le istanze nel database seguiranno la nomenclatura: doRemove**NomeDellaClasse**
* I metodi del DAO per cancellare le istanze nel database secondo un determinato parametro seguiranno la nomenclatura: doRemove**NomeDellaClasse**By**Parametro**

## Definizioni, Acronimi e Abbreviazioni

|  |  |
| --- | --- |
| **Acronimo/Abbreviazione** | **Definizione** |
| Package | Raggruppamento di classi, interfacce o file correlati |
| Design pattern | Template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità |
| Interfaccia | Insieme di signature delle operazioni offerte dalla classe |
| View | Nel pattern MVC rappresenta ciò che viene visualizzato a schermo da un utente e che gli permette di interagire con le funzionalità offerte dalla piattaforma |
| lowerCamelCase | E’ la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione nel mezzo della frase inizi con una lettera maiuscola, senza spazi o punteggiatura intermedi |
| UpperCamelCase | E’ la pratica di scrivere frasi in modo tale che ogni parola o abbreviazione inizi con una lettera maiuscola, senza spazi o punteggiatura intermedi |
| Javadoc | Sistema di documentazione offerto da Java, che viene generato sottoforma di interfaccia in modo da rendere la documentazione accessibile e facilmente leggibile. |

## Riferimenti

Di seguito una lista di riferimenti ad altri documenti utili durante la lettura:

1. Packages

In questa sezione viene mostrata la suddivisione del sistema in package, in base a quanto definito nel documento di System Design. Tale suddivisione è motivata dalle scelte architetturali prese.

**Package SalernArte**

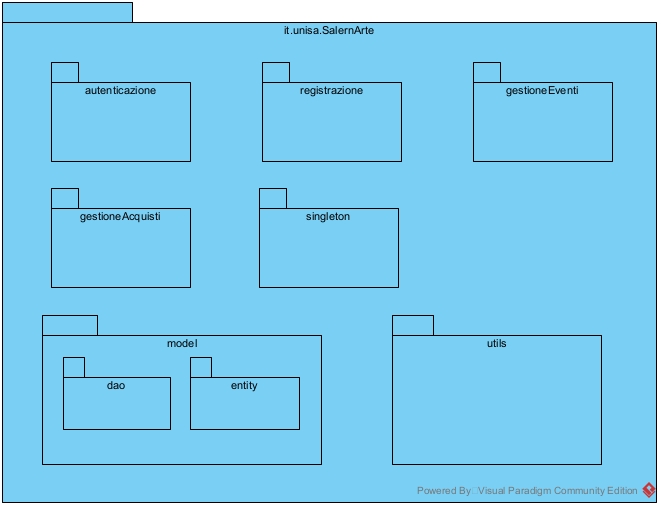
Nella presente sezione si mostra la struttura del package pricipale di SalernArte. La struttura generale è stata ottenuta a partire da tre principali scelte:

1. Creare un package separato per ogni sottosistema, contenente le classi service e controller del sottosistema, ed eventuali classi di utilità usate unicamente da esso.

2. Creare un package separato per le classi del *model*, contenente le classi entity e i DAO per l’accesso al DB. Tale scelta è stata presa vista l’elevata complessità del database di SalernArte che prevede numerose relazioni tra le entità. Si è quindi preferito tenere tutto in un package separato e collegato a tutti gli altri package dei sottosistemi.

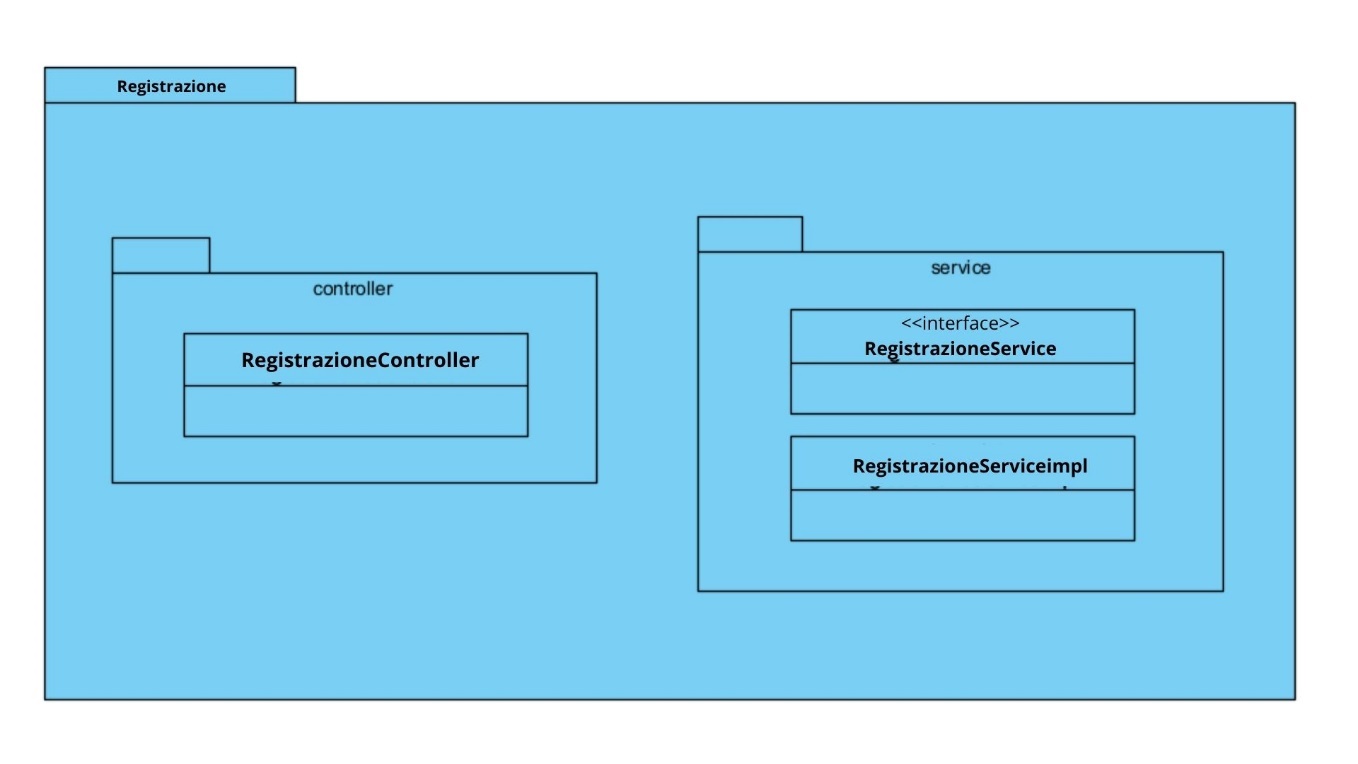
3. Creare un package chiamato *utils* in cui inserire eventuli classi di utlità per il sistema e usabili da più sottosistemi.

Per ciò che concerne la dipendenza tra i packages, la suddivisione precedentemente illustrata a portato alla creazione di una relazione tra il package model e tutti gli altri package del sistema.

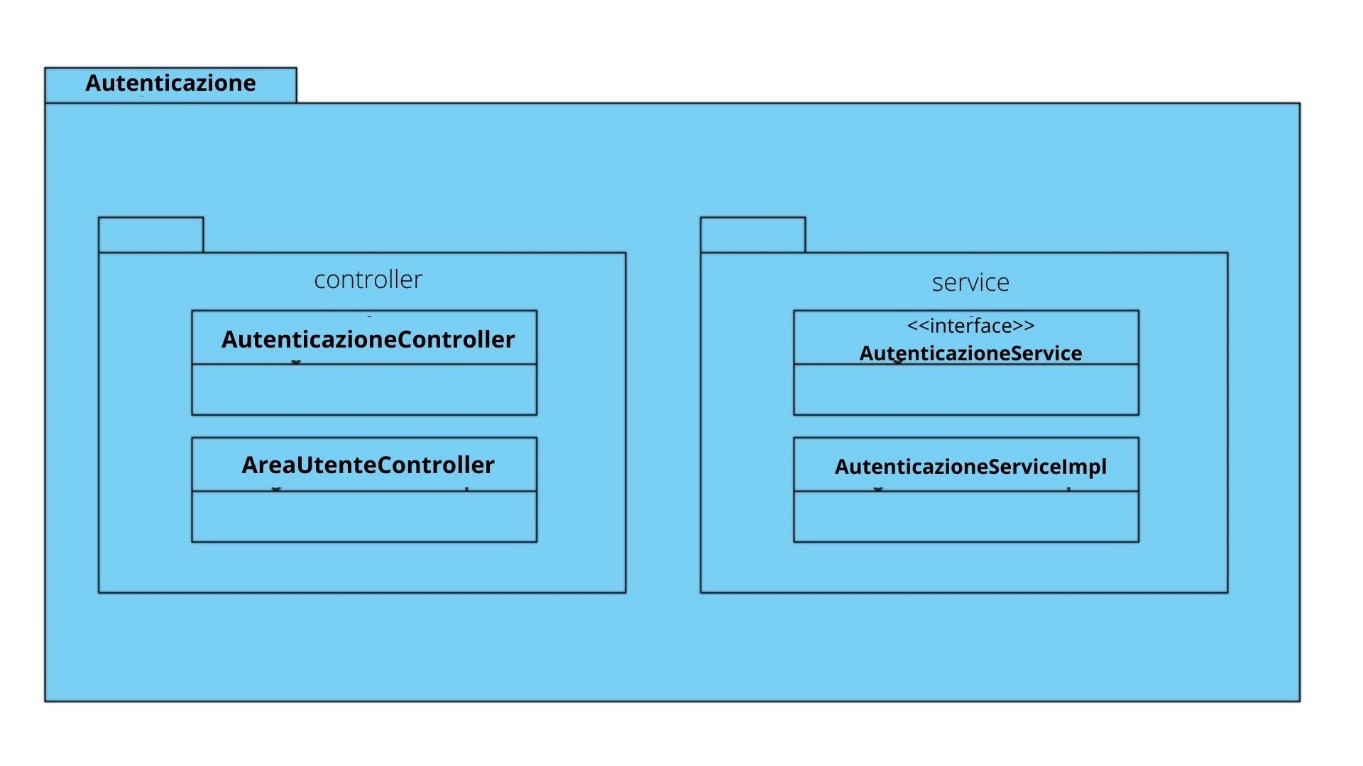


**SECONDO ME A SITO CREATO VANNO RAPPRESENTATI ANCHE I PACKAGE DI MODEL E UTILS NEL DETTAGLIO**

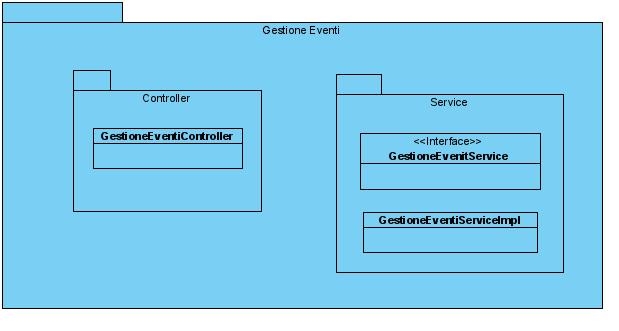
**Package Registrazione**



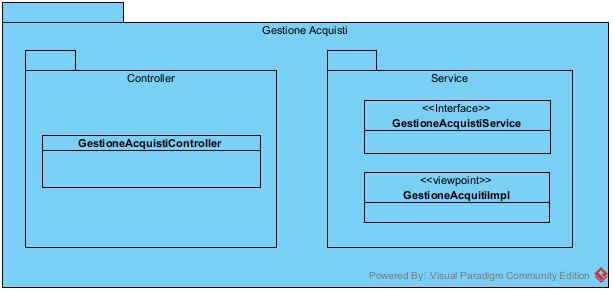
**Package Autenticazione**



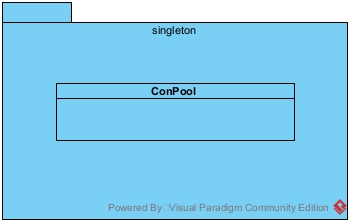
**Package GestioneEventi**

****

**Package Gestione Acquisti**



**Package Singleton**



1. Class Interfaces

Di seguito saranno presentate le interfacce di ciascun package:

**Javadoc di Salernarte**

Per motivi di leggibilità di è scelto di creare un sito, hostato tramite GitHub pages, contenente la JavaDoc di Salernarte. In questo modo, chiunque può consultare la documentazione aggiornata dell’intero sistema.

Di seguito il link al sito in questione:

**Package Registrazione**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | | **Autore** | **ML** | |
| **Nome Classe** | | **RegistrazioneService** | | | |
| **Descrizione** | | Questa classe permette di gestire le operazioni relative alla registrazione | | | |
| **Metodi** | | +registraOspite(Ospite ospite): UtenteRegistrato  +registraScolaresca(Scolaresca scolaresca): UtenteRegistrato  + registraAdmin(Amministratore amministratore): UtenteRegistrato | | | |
| **Invariante di classe** | | / | | | |
| **Nome Metodo** | +registraOspite(Ospite ospite): UtenteRegistrato | | | |
| **Descrizione** | Questo metodo consente la registrazione di un ospite | | | |
| **Pre-condizione** | / | | | |
| **Post-condizione** | context: RegistrazioneService:: registraOspite(Ospite ospite)  post: OspiteDAO.save(ospite)==true | | | |
| **Nome Metodo** | +registraScolaresca(Scolaresca scolaresca): UtenteRegistrato | | | |
| **Descrizione** | Questo metodo consente la registrazione di una scolaresca | | | |
| **Pre-condizione** | / | | | |
| **Post-condizione** | context: RegistrazioneService:: registraScolaresca(Scolaresca scolaresca)  post: ScolarescaDAO.save(ospite)==true | | | |
| **Nome Metodo** | +registraAdmin(Amministratore amministratore): UtenteRegistrato | | | |
| **Descrizione** | Questo metodo consente la registrazione di un admin | | | |
| **Pre-condizione** | / | | | |
| **Post-condizione** | **context:** RegistrazioneService:: registraAdmin(Amministratore amministratore)  **post:** AmministratoreDAO.save(ospite)==true | | | |

**Package Autenticazione**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **Autore** | **ML** |
| **Nome Classe** | **AutenticazioneService** | | |
| **Descrizione** | Questa classe permette di gestire le operazioni relative all’autenticazione | | |
| **Metodi** | +login(String email, String password): UtenteRegistrato  +aggiornaUtenteRegistrato(UtenteRegistrato utenteRegistrato):UtenteRegistrato  +aggiornaScolaresca(Scolaresca scolaresca):Scolaresca  +aggiornaAdmin(Amministratore amministratore):Admin | | |
| **Invariante di classe** | / | | |

|  |  |
| --- | --- |
| **Nome Metodo** | +login(String email, String password) |
| **Descrizione** | Questo metodo consente di loggare un utente registrato |
| **Pre-condizione** | / |
| **Post-condizione** | **context:** AutenticazioneService::login(email,password) |
| **Nome Metodo** | +aggiornaUtenteRegistrato(UtenteRegistrato utenteRegistrato) |
| **Descrizione** | Questo metodo consente di aggiornare i dati di un utente registrato |
| **Pre-condizione** | / |
| **Post-condizione** | / |
| **Nome Metodo** | +aggiornaScolaresca(Scolaresca scolaresca) |
| **Descrizione** | Questo metodo consente di aggiornare i dati di una scolaresca |
| **Pre-condizione** | / |
| **Post-condizione** | / |
| **Nome Metodo** | +aggiornaAdmin(Amministratore amministratore) |
| **Descrizione** | Questo metodo consente di aggiornare i dati di un amministratore |
| **Pre-condizione** | / |
| **Post-condizione** | / |

**Package Gestione Eventi**

**(conviene secondo me sempre farli dopo design pattern ed implementazione, ma…)**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **Autore** | **DPA** |
| **Nome Classe** | **GestioneEventiService** | | |
| **Descrizione** | Questa classe permette di gestire le operazioni relative alla gestione degli eventi | | |
| **Metodi** | +VisualizzaListaEventiTeatrali( ) : List<Evento>  +VisualizzaListaEventiMostre( ) : List<Evento>  +VisualizzaEvento( int id\_evento): Evento  +VisualizzaBioOrg(int id\_organizzatore): String  +RicercaEvento (String nome\_evento): List<Evento>  +VisualizzaEventiAdmin( ): List<Evento>  +VisualizzaRichiesteOrganizzatore(): List<Evento> FLAG IN EVENTO PER LE MODIFICHE / ATTIVO OP MENO  +AccettaRichiestaEvento(int id\_evento):void  +RifiutaRichiestaEvento(int id\_evento):void  +VisualizzaEventiOrg( int id\_organizzatore):List<Evento>  +RichiestaEvento(Evento evento): Evento  +RichiestaModificaEvento(Evento evento): Evento  +EliminaEvento(int id\_evento): void | | |
| **Invariante di classe** | / | | |

|  |  |
| --- | --- |
| **Nome Metodo** | **+VisualizzaListaEventiTeatrali( ) : List<Evento>** |
| **Descrizione** | Questo metodo consente di visualizzare la lista degli eventi teatrali attivi nel sistema e restituisce una lista contenente tali eventi |
| **Pre-condizione** | / |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+VisualizzaListaEventiMostre( ) : List<Evento>** |
| **Descrizione** | Questo metodo consente di visualizzare la lista degli eventi di tipo mostra attivi nel sistema e restituisce una lista contenente tali eventi |
| **Pre-condizione** | / |
| **Post-condizione** | / |
| **Nome Metodo** | **+VisualizzaEvento( int id\_evento): Evento** |
| **Descrizione** |  |
| **Pre-condizione** | / |
| **Post-condizione** | / |
| **Nome Metodo** | **+VisualizzaBioOrg(int id\_organizzatore): String** |
| **Descrizione** |  |
| **Pre-condizione** | / |
| **Post-condizione** | / |
| **Nome Metodo** | **+RicercaEvento (String nome\_evento): List<Evento>** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+VisualizzaEventiAdmin( ): List<Evento>** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+VisualizzaRichiesteOrganizzatore(): List<Evento>** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+AccettaRichiestaEvento(int id\_evento):void** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+RifiutaRichiestaEvento(int id\_evento):void** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+VisualizzaEventiOrg( int id\_organizzatore):List<Evento>** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+RichiestaEvento(Evento evento): Evento** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+RichiestaModificaEvento(Evento evento): Evento** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |
| **Nome Metodo** | **+EliminaEvento(int id\_evento): void** |
| **Descrizione** |  |
| **Pre-condizione** | **/** |
| **Post-condizione** | **/** |

**Package Gestione Acquisti**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **Autore** | **LM** |
| **Nome Classe** | **GestioneAcquistiService** | | |
| **Descrizione** | Questa classe permette di gestire le operazioni riguardanti gli acquisti degli eventi | | |
| **Metodi** | +acquistaBiglietti(List<Evento> listaeventi, Integer id): UtenteRegistrato  +modificaCarrello(Integer quantità, Integer idEvento): UtenteRegistrato | | |
| **Invariante di classe** | / | | |

|  |  |
| --- | --- |
| **Nome Metodo** | +acquistaBiglietti(List<Evento> listaeventi, Integer id) |
| **Descrizione** | Questo metodo permette di acquistare gli eventi nel carrello con la relativa quantità di ogni evento. |
| **Pre-condizione** | listaeventi.get(i).getQuantità > 0 |
| **Post-condizione** | setOrdersAccount(Integer id, Ordine o)  “lista ordini account +1; quantità di ogni evento iniziale – quantità di ogni evento acquistato”. |

|  |  |
| --- | --- |
| **Nome Metodo** | + modificaCarrello(Integer quantità, Integer idEvento) |
| **Descrizione** | Questo metodo permette di modificare la quantità di ogni singolo evento del carrello |
| **Pre-condizione** | listaeventi.get(i).getQuantità > 0 &&  quantità < listaeventi.get(i).getQuantità |
| **Post-condizione** | / |

3 Class Diagram

## 3.1 Package model

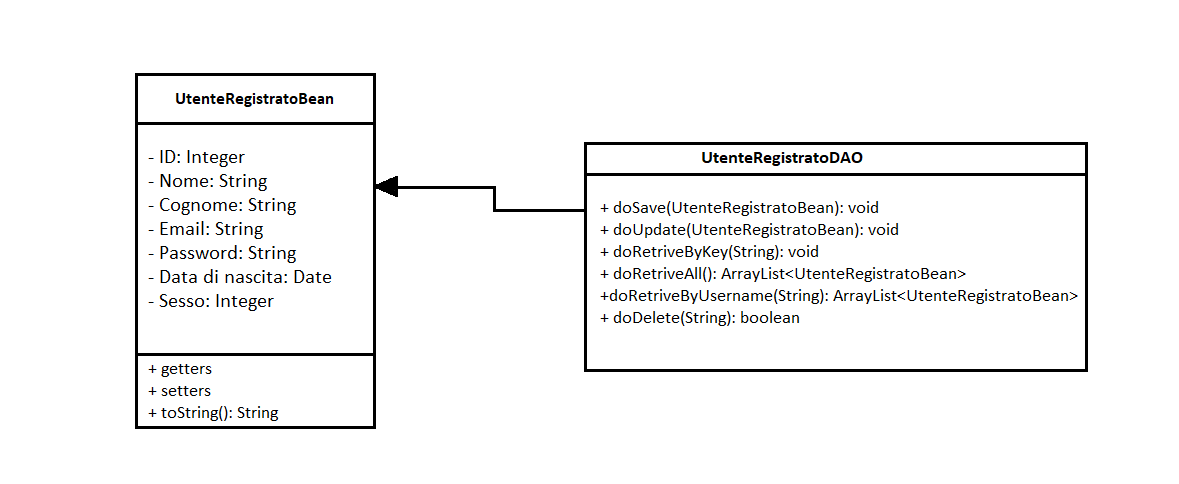


Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

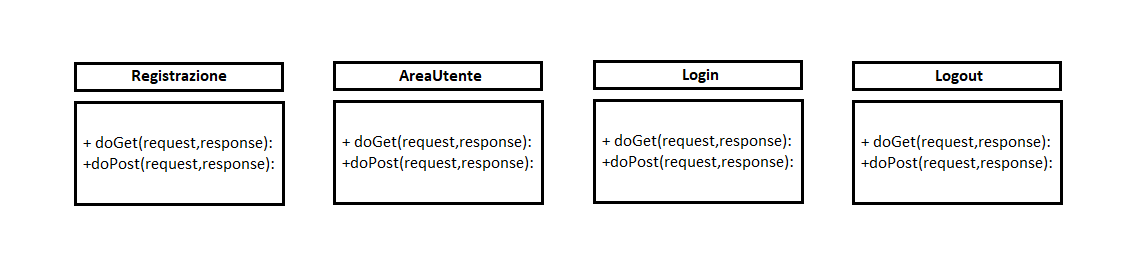
Immagine che contiene testo

Descrizione generata automaticamente

Immagine che contiene testo

Descrizione generata automaticamente

## 3.2 Package control



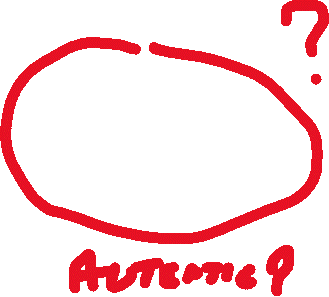


Immagine che contiene testo

Descrizione generata automaticamente





Capisco che servano servlet diverse per login e logout però mi confonde perché prima abbiamo parlato di autenticazioneController ed adesso di login e logout

4 Design Patterns

Il primo design pattern scelto è il **Façade***.* Il pattern è basato sull’utilizzo di una classe che consente, tramite un’interfaccia più semplice, l’accesso ad un sottosistema che contiene classi con interfacce complesse e diverse tra loro. Questo design pattern offre due vantaggi:

* La riduzione del numero di associazioni
* Semplicità nell’attuazione di cambiamenti

Ogni package è caratterizzato da un sub-package chiamato “services”, contenente le classi che implementano i metodi corrispondenti ai servizi offerti dal sottosistema. Questi metodi usufruiranno delle classi situate nei package “model” e “dao”. Attraverso questa strutturazione dei package, si ottiene una separazione tra logica di business e logica di controllo.

Di seguito è mostrato un esempio d’uso del design pattern *Façade.*

Il diagramma mostra il collegamento tra Controller e Services grazie all’utilizzo del suddetto design pattern.



Il secondo design pattern individuato è il **Singleton**, che appartiene alla categoria dei design pattern “creazionali” e ha lo scopo di garantire che una classe possa essere istanziata una sola volta, e di fornire un punto di accesso globale a tale istanza. Inoltre, integra due nuove funzionalità:

* Ha un costruttore privato
* Definisce un operatore public static che viene impiegato per accedere al singleton

Questo design pattern verrà utilizzato nel nostro sistema, per permettere la connessione al database da parte delle classi situate nei package “dao”.



Il terzo e ultimo design pattern individuato è il **DAO,** utilizzato per gestire la logica di accesso ai dati persistenti. Di seguito è mostrato un esempio d’uso del design pattern DAO.

Il diagramma illustra come il suddetto design pattern nasconda alla ClasseService le implementazioni delle operazioni di accesso ai dati persistenti, implementando il concetto di “loosely coupled”.



5 Glossario

|  |  |
| --- | --- |
| **Sigla/Termine** | **Definizione** |
| **Package** | Raggruppamento di classi e interfacce |
| **DAO** | Data Access Object, design pattern architetturale che si occupa di fornire in modo astratto l’accesso ai dati persistenti |
| **Controller** | Classe che gestisce le richieste effettuate dal client |
| **Service** | Classe che implementa la logica di business, utilizzata dal controller o altro sottosistema |
| **Model** | Parte dell’MVC che fornisce i metodi per accedere ai dati utili al sistema |
| **MVC** | Model View Controller, design architetturale che separa la logica di presentazione dalla logica di business |
| **Facade** | Classe che utilizza un’interfaccia semplice per accedere ai metodi di un’interfaccia complessa |
| **Singleton** | Design pattern creazione con lo scopo di strutturare una sola istanza di una classe fornendo un punto di accesso globale |