



**Relazione finale del progetto  
del corso di  
Progettazione e Algoritmi**

**Progettazione di un sistema di gestione per associazione musicale**

Studenti:

Boffelli Jacopo                      1053217

Panzeri Federico                      1052952

*Febbraio 2022*

Laurea Magistrale in Ingegneria Informatica

Università degli Studi di Bergamo

A.A 2021/2022



# Sommario

<b>ITERAZIONE 0</b>	<b>5</b>
<b>Documento requisiti software</b>	<b>5</b>
<b>Scopo</b>	<b>5</b>
<b>Definizione dei requisiti</b>	<b>5</b>
<b>Requirements envisioning</b>	<b>7</b>
Descrizione testuale dei use-case diagram	7
Casi d'uso ad alta priorità	7
Casi d'uso a media priorità	8
Casi d'uso a bassa priorità	8
<b>Use case diagram</b>	<b>9</b>
<b>Topologia Architetturale</b>	<b>10</b>
<b>Toolchain</b>	<b>12</b>
<b>ITERAZIONE 1</b>	<b>13</b>
<b>UML Component Diagram</b>	<b>13</b>
<b>UML - Class Diagram per interfacce</b>	<b>15</b>
<b>UML - Class Diagram per tipi di dato nelle interfacce (presentation model)</b>	<b>15</b>
<b>Modello E/R</b>	<b>16</b>
<b>ITERAZIONE 2</b>	<b>17</b>
<b>Descrizione dei casi d'uso.</b>	<b>18</b>
UC1: Gestione Anagrafiche Soci	18
UC2: Gestione Eventi	20
<b>UML - Component Diagram (componenti implementate)</b>	<b>23</b>
<b>Class Diagram dei metodi</b>	<b>24</b>
<b>Tecnologie utilizzate</b>	<b>25</b>
<b>Testing API tramite PostMan</b>	<b>26</b>
<b>Analisi Dinamica</b>	<b>27</b>
<b>ITERAZIONE 3</b>	<b>30</b>
<b>Descrizione del caso d'uso.</b>	<b>30</b>
UC3: Gestione Organico relativo ad un certo Evento	30
UC4: Algoritmo Gestione Organico	31
<b>Algoritmo</b>	<b>32</b>
Passo 1	32
Passo 2	32
Passo 3	32
Costo Computazionale	33
<b>UML - Component Diagram delle componenti implementate</b>	<b>35</b>
<b>Class Diagram dei metodi</b>	<b>35</b>
<b>UML Sequence Diagram</b>	<b>36</b>

<b>Test chiamata GET Algoritmo</b>	<b>37</b>
<b>Analisi dinamica</b>	<b>38</b>
<b>Analisi statica del codice (CodeMR)</b>	<b>39</b>
Distance: legame tra Abstractness e Instability	39
Treemap View	40
Grafo strutturale	41
Project Outline	42

# ITERAZIONE 0

## Documento requisiti software

### Scopo

Creare un applicativo software distribuito per la gestione degli eventi e delle risorse di un'associazione musicale dilettantistica chiamato “*eBand*”.

Questo progetto parte dal presupposto che un'associazione musicale debba gestire cinque macrorisorse:

1. Soci
2. Eventi
3. Strumenti
4. Brani
5. Vestiario

Si ipotizza quindi che tutte e cinque le categorie indicate siano necessarie al buon funzionamento delle attività del gruppo musicale; tuttavia il focus di questo progetto sarà la gestione e l'organizzazione efficiente dei soci al fine di portare a termine più eventi possibili e per questo l'implementazione della parte di software gestionale si limiterà ai soli punti 1 e 2.

### Definizione dei requisiti

Il Sistema consente agli amministratori di sovrintendere gli aspetti principali della gestione delle attività dell'associazione musicale. A tal scopo dispone di diverse interfacce per la gestione dei soci, degli eventi e dell'inventario.

Nel dettaglio, inserendo nome utente e password per effettuare il login, gli amministratori possono:

- Accedere ad un'area riservata (anagrafica) per visualizzare, aggiungere, modificare e rimuovere i dati di un socio
- Accedere ad un'area riservata (eventi) per visualizzare, aggiungere, modificare e rimuovere un evento
- Accedere ad un'area riservata (inventario) per visualizzare, aggiungere, modificare e rimuovere elementi dell'inventario, nello specifico strumenti musicali, vestiario e brani
- Accedere ad un'area riservata per visualizzare la formazione, intesa come composizione dell'organico strumentale, calcolata dall'algoritmo per un determinato evento sulla base delle presenze
- Accedere ad un'area riservata (rubrica) per visualizzare, aggiungere, modificare e rimuovere i dati di contatto.
- Eseguire il logout

Nel dettaglio, inserendo nome utente e password per effettuare il login, i soci possono:

- Accedere ad un'area riservata per visualizzare le informazioni (data, ora, luogo, tipologia) relative ad un determinato evento (a prescindere che vi partecipino o meno) ed eventualmente confermare la propria presenza o assenza.
- Eseguire il logout.

## Requirements envisioning

Al fine di procedere ad uno sviluppo efficiente, è stato deciso di dividere le specifiche funzionali in tre code di priorità: alta, media e bassa.

Nella coda a priorità alta si troveranno tutti i casi d'uso necessari al buon funzionamento dell'applicazione, beninteso che lo svolgimento di questo elaborato altro non è che un esercizio didattico volto ad integrare attività pratiche di scrittura di software con attività più teoriche di progettazione, sia architetturale che algoritmica.

Nella coda a media priorità saranno inseriti i casi d'uso riguardanti funzionalità aggiuntive e nella coda a bassa priorità le funzionalità cosiddette "Nice-To-Have" che verranno implementate in versioni future.

## Descrizione testuale dei casi d'uso

Casi d'uso ad alta priorità

CODICE	DESCRIZIONE
UC1	Gestione Anagrafiche Soci (visualizzazione, inserimento, modifica, cancellazione)
UC2	Gestione Eventi (visualizzazione, inserimento, modifica, cancellazione)
UC3	Gestione Organico relativo ad un certo evento (visualizzazione)
UC4	Algoritmo gestione organico

### Casi d'uso a media priorità

CODICE	DESCRIZIONE
UC5	Login Utente
UC6	Logout Utente
UC7	Visualizzazione eventi (utente)
UC8	Conferma presenza evento (utente)

### Casi d'uso a bassa priorità

UC9	Gestione Inventario Strumenti (visualizzazione, inserimento, modifica, cancellazione)
UC10	Gestione Vestiario (visualizzazione, inserimento, modifica, cancellazione)
UC11	Gestione Brani (visualizzazione, inserimento, modifica, cancellazione)
UC12	Gestione Rubrica (visualizzazione, inserimento, modifica, cancellazione)



## Use case diagram

Di seguito lo schema UML dei casi d'uso.

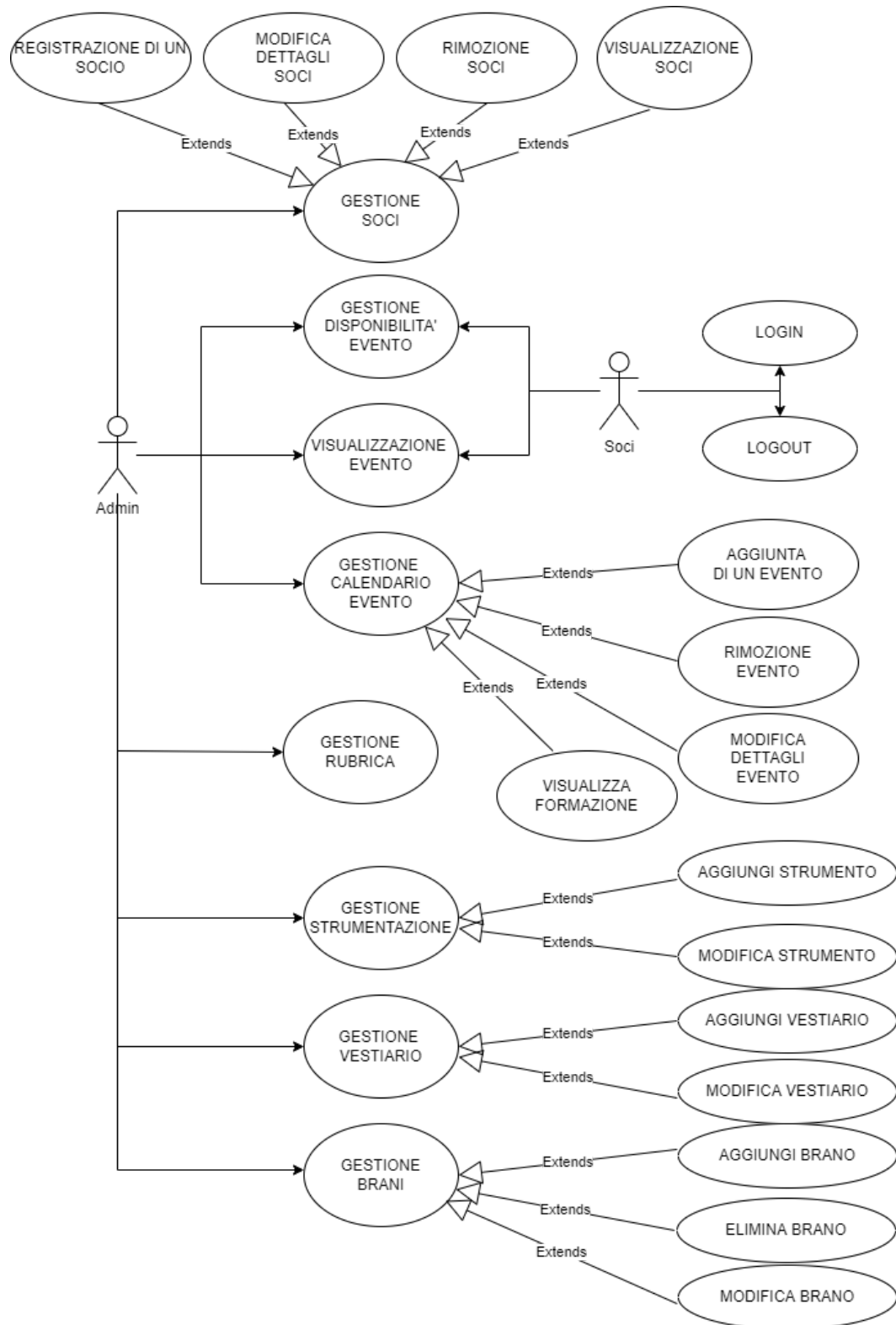


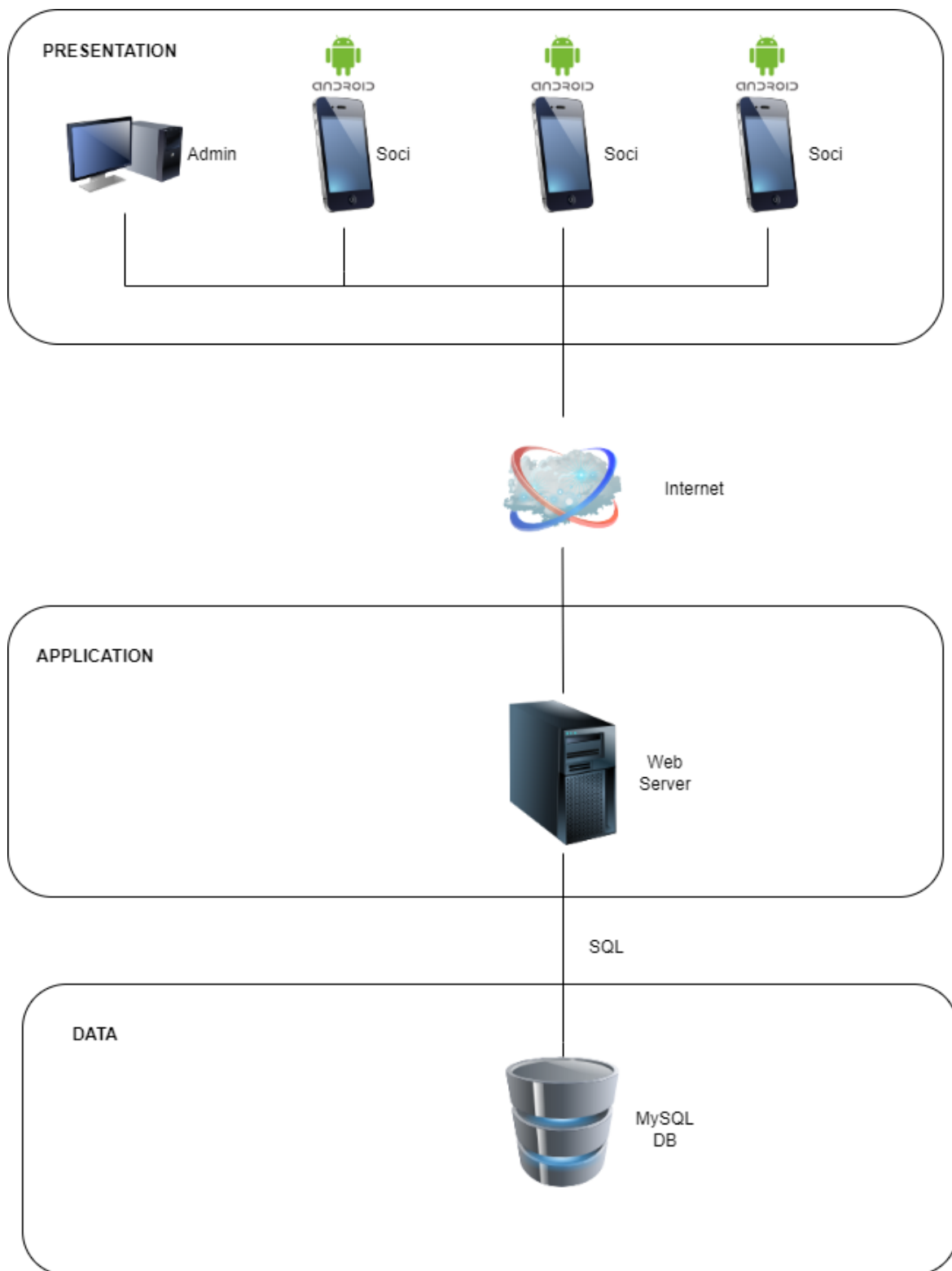
Figura 1 - Diagramma UML casi d'uso

## Topologia Architetture

Facendo riferimento alla figura seguente (figura 2), il sistema è costituito da tre livelli (architettura 3-tier) così denominati:

- *Application*, per quanto riguarda la logica funzionale
- *Presentation*, per quanto concerne l'interfaccia utente
- *Data*, relativamente alla gestione dei dati persistenti.

In particolar modo, il livello *Data* ospita il database relazionale realizzato tramite MySQL; *Application* contiene la logica funzionale implementata in Java, la connessione con il db (gestita dal framework *Spring*) ed espone delle API per consentire a *Presentation* di recuperare i dati già elaborati da mostrare agli utenti o tramite la WebApp scritta in React.js o tramite l'applicazione Android.



*Figura 2 - Topologia di sistema*

# Toolchain

Elenco:

## 1 Modellazione:

- Use case diagram, deployment diagram, class diagram, component diagram:  
*Draw.io e Notability.*

## 2 Implementazione Software

- Linguaggio di programmazione: *Java, JavaScript, Html*
- IDE: *Eclipse e Visual Studio*
- API Development: *Eclipse e Postman*
- DBMS: *MySQL*
- Interfaccia grafica: *React.js*

## 3 Analisi del Software

- Analisi Statica: *CodeMR*
- Analisi Dinamica: *JUnit*

## 4 Documentazione, Versioning e gestione del team

- Versioning: *Git* e *GitHub*
- Gestione Team: *Google Drive, Google Meet, Trello*

# ITERAZIONE 1

Durante questa iterazione il team si è concentrato principalmente sullo sviluppo dell'intera architettura software. L'obiettivo era quello di focalizzare l'attenzione sulla struttura del sistema intero e di realizzare diverse rappresentazioni che potessero agevolarne la comprensione, nonché di ipotizzare la struttura del database e rappresentare di conseguenza il diagramma entità-relazione.

## **UML Component Diagram**

Il seguente diagramma dei componenti rappresenta tutti gli elementi software principali del sistema ponendo attenzione su come essi interagiscono tramite la notazione ball and socket, quest'ultima infatti risalta quale componente espone l'interfaccia e quale ne usufruisce.

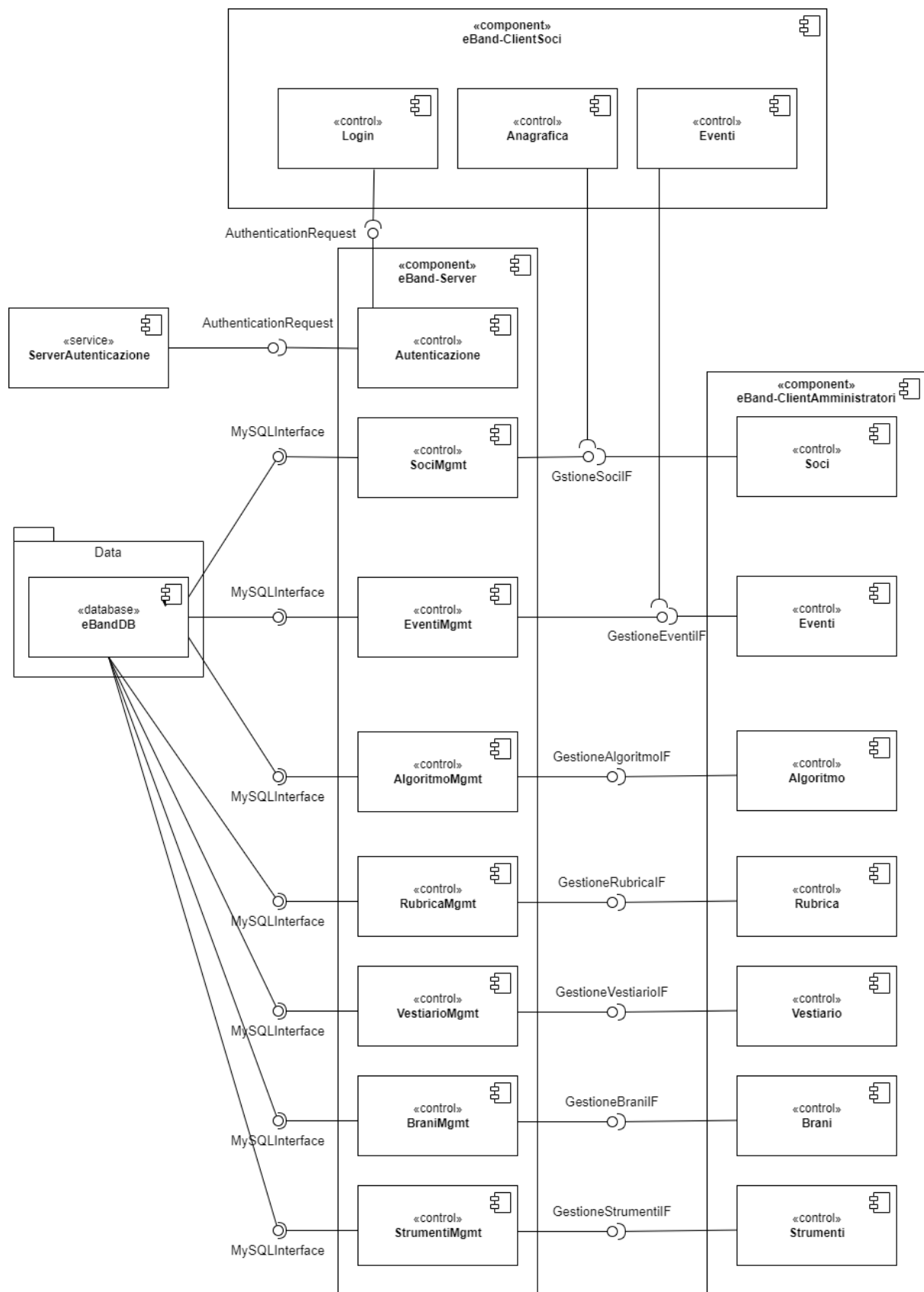


Figura 3 - Component Diagram

## UML - Class Diagram per interfacce

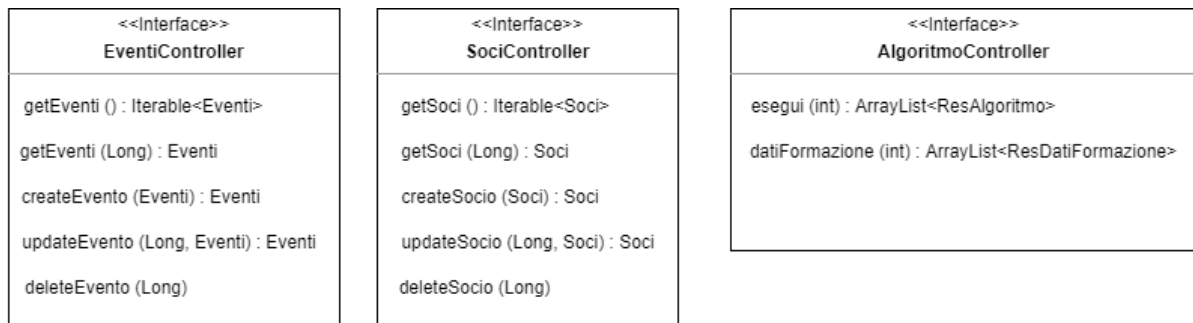


Figura 4 - Class Diagram per interfacce

## UML - Class Diagram per tipi di dato nelle interfacce (presentation model)

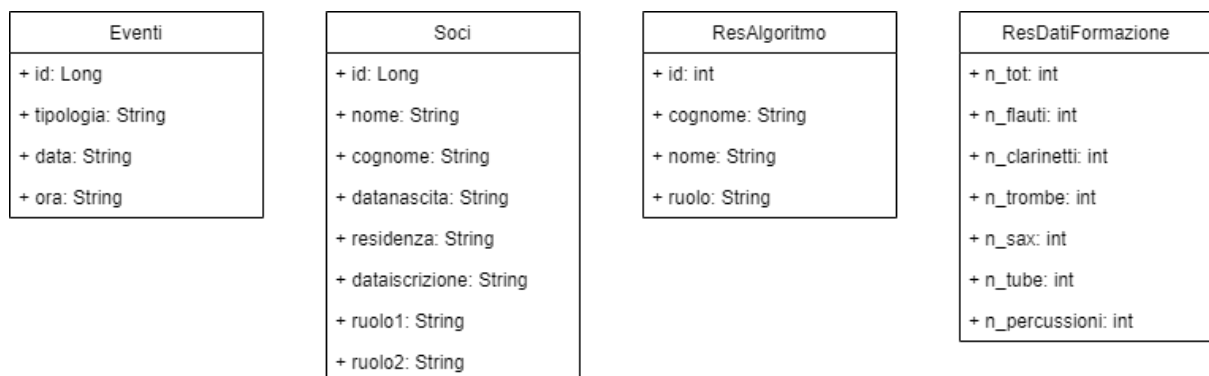


Figura 5 - Class Diagram per tipi di dato

## Modello E/R

Con il seguente diagramma mostriamo la progettazione del database. Vengono mostrate tutte le entità (classi di oggetti) con i relativi attributi e le relazioni che rappresentano il legame tra le varie entità.

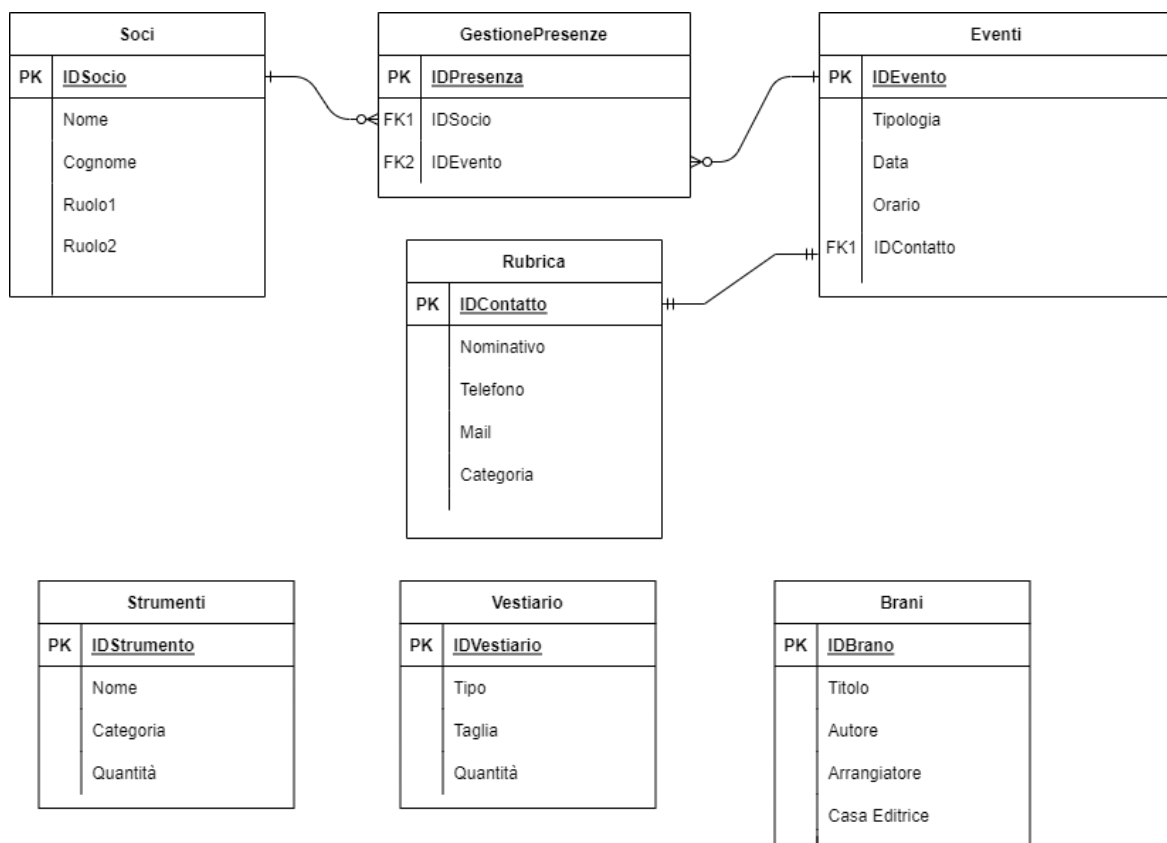


Figura 6 - Diagramma entità-relazione del database



## ITERAZIONE 2

Nell'iterazione 2 si è passati dalla progettazione architetturale all'implementazione vera e propria dello scheletro dell'applicazione, ossia delle funzionalità di base inserite nella coda "ad alta priorità" dell'iterazione 0.

Per farlo è stato necessario raggruppare le funzionalità principali alla base dell'applicazione in macro categorie di casi d'uso.

Di seguito quelli implementati nell'iterazione 2:

- UC1: Gestione Anagrafiche Soci
  - UC1.1: Visualizzazione
  - UC1.2: Inserimento
  - UC1.3: Modifica
  - UC1.4: Cancellazione
  
- UC2: Gestione Eventi
  - UC2.1: Visualizzazione
  - UC2.2: Inserimento
  - UC2.3: Modifica
  - UC2.4: Cancellazione

## **Descrizione dei casi d'uso.**

### **UC1: Gestione Anagrafiche Soci**

*Breve descrizione:* l'amministratore deve avere una visione generale dei soci che fanno parte dell'associazione. Perciò, oltre alla visualizzazione, deve poter inserire un nuovo socio quando qualcuno si iscrive all'associazione, modificare le relative informazioni qualora presentino degli errori o siano cambiate (es: cambio di residenza) ed infine deve poterlo eliminare qualora decida di lasciare l'organizzazione (in ottemperanza alle leggi sulla privacy).

Per questo la gestione dei soci è suddivisa in quattro casi d'uso concreti:

- UC1.1: Visualizzazione
- UC1.2: Inserimento
- UC1.3: Modifica
- UC1.4: Cancellazione

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore accede alla homepage dell'applicazione e il Sistema mostra un header con i pulsanti "Home", "Eventi", "Soci", "EventOptimizer".

L'amministratore clicca su "Soci" e il sistema mostra a schermo la pagina relativa alla gestione delle informazioni relative ai soci

#### **UC1.1: Visualizzazione Anagrafiche Soci**

*Breve descrizione:* l'amministratore visualizza i soci inseriti a sistema e le relative informazioni

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore accede alla homepage dell'applicazione e il Sistema mostra un header con i pulsanti "Home", "Eventi", "Soci", "EventOptimizer".

L'amministratore clicca su "Soci" e il sistema mostra a schermo la pagina relativa alla gestione delle informazioni relative ai soci. Il primo campo della pagina é una lista che visualizza nome e cognome dei vari soci iscritti, nonché tutte le relative informazioni

### **UC1.2: Inserimento Anagrafiche Soci**

*Breve descrizione:* l'amministratore inserisce un nuovo socio a sistema con le relative informazioni. Questa operazione viene fatta principalmente nella fase di setup iniziale dell'applicazione e poi in seguito all'iscrizione di nuovi soci.

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore si trova nella pagina "Soci" e naviga fino al form dedicato all'inserimento di un nuovo socio, lo compila e preme il tasto "Inserisci"

### **UC1.3: Modifica Anagrafiche Soci**

*Breve descrizione:* l'amministratore modifica le informazioni associate ad un socio. Questa operazione viene resa disponibile per soddisfare quei casi in cui sono state inserite delle informazioni errate oppure sono cambiate alcune informazioni relative ad un socio.

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore si trova nella pagina "Soci" e naviga fino al form dedicato alla modifica di un socio, lo compila e preme il tasto "Aggiorna"

#### **UC1.4: Cancellazione Anagrafiche Soci**

*Breve descrizione:* l'amministratore cancella tutte le informazioni di un socio.

Questa operazione viene resa disponibile per soddisfare quei casi in cui una persona lascia l'associazione oppure decede..

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore si trova nella pagina "Soci" e naviga fino al form dedicato alla cancellazione di un socio, lo compila e preme il tasto "Rimuovi dal Database"

#### **UC2: Gestione Eventi**

*Breve descrizione:* l'amministratore deve avere una visione generale degli eventi a cui l'associazione vuole prendere parte. Perciò, oltre alla visualizzazione, deve poter inserire un nuovo evento quando viene confermato, modificare le relative informazioni qualora presentino degli errori o siano cambiate ed infine deve poterlo eliminare qualora l'evento sia passato.

Per questo la gestione degli eventi è suddivisa in quattro casi d'uso concreti:

- UC2.1: Visualizzazione
- UC2.2: Inserimento
- UC2.3: Modifica
- UC2.4: Cancellazione

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore accede alla homepage dell'applicazione e il Sistema mostra un header con i pulsanti "Home", "Eventi", "Soci", "EventOptimizer".

L'amministratore clicca su "Eventi" e il sistema mostra a schermo la pagina relativa alla gestione delle informazioni relative ai soci

### **UC2.1: Visualizzazione Eventi**

*Breve descrizione:* l'amministratore visualizza gli eventi inseriti a sistema e le relative informazioni

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore accede alla homepage dell'applicazione e il Sistema mostra un header con i pulsanti "Home", "Eventi", "Soci", "EventOptimizer".

L'amministratore clicca su "Eventi" e il sistema mostra a schermo la pagina relativa alla gestione delle informazioni relative agli eventi. Il primo campo della pagina é una lista che visualizza, tra le varie cose, Tipologia, Data e Ora di tutti gli eventi disponibili a DB.

### **UC2.2: Inserimento Nuovo Evento**

*Breve descrizione:* l'amministratore inserisce un nuovo evento a sistema con le relative informazioni. Questa operazione viene fatta ogni volta che l'associazione conferma la propria presenza ad un determinato evento.

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore si trova nella pagina "Eventi" e naviga fino al form dedicato all'inserimento di un nuovo evento, lo compila e preme il tasto "Inserisci"

### **UC2.3: Modifica Evento**

*Breve descrizione:* l'amministratore modifica le informazioni associate ad un evento. Questa operazione viene resa disponibile per soddisfare quei casi in cui sono state inserite delle informazioni errate oppure sono cambiate alcune informazioni relative all'evento.

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore si trova nella pagina "Eventi" e naviga fino al form dedicato alla modifica di un evento, lo compila e preme il tasto "Aggiorna"

### **UC2.4: Cancellazione Evento**

*Breve descrizione:* l'amministratore rimuove un evento dal sistema. Questa operazione viene resa disponibile per rimuovere gli eventi passati

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore si trova nella pagina "Soci" e naviga fino al form dedicato alla cancellazione di un socio, lo compila e preme il tasto "Rimuovi dal Database"

## UML - Component Diagram (componenti implementate)

Le componenti che implementano le funzionalità scelte per l'implementazione nell'iterazione 2 sono quelle descritte nel *component diagram* seguente che evidenzia le interazioni tra i componenti dal punto di vista della funzionalità.

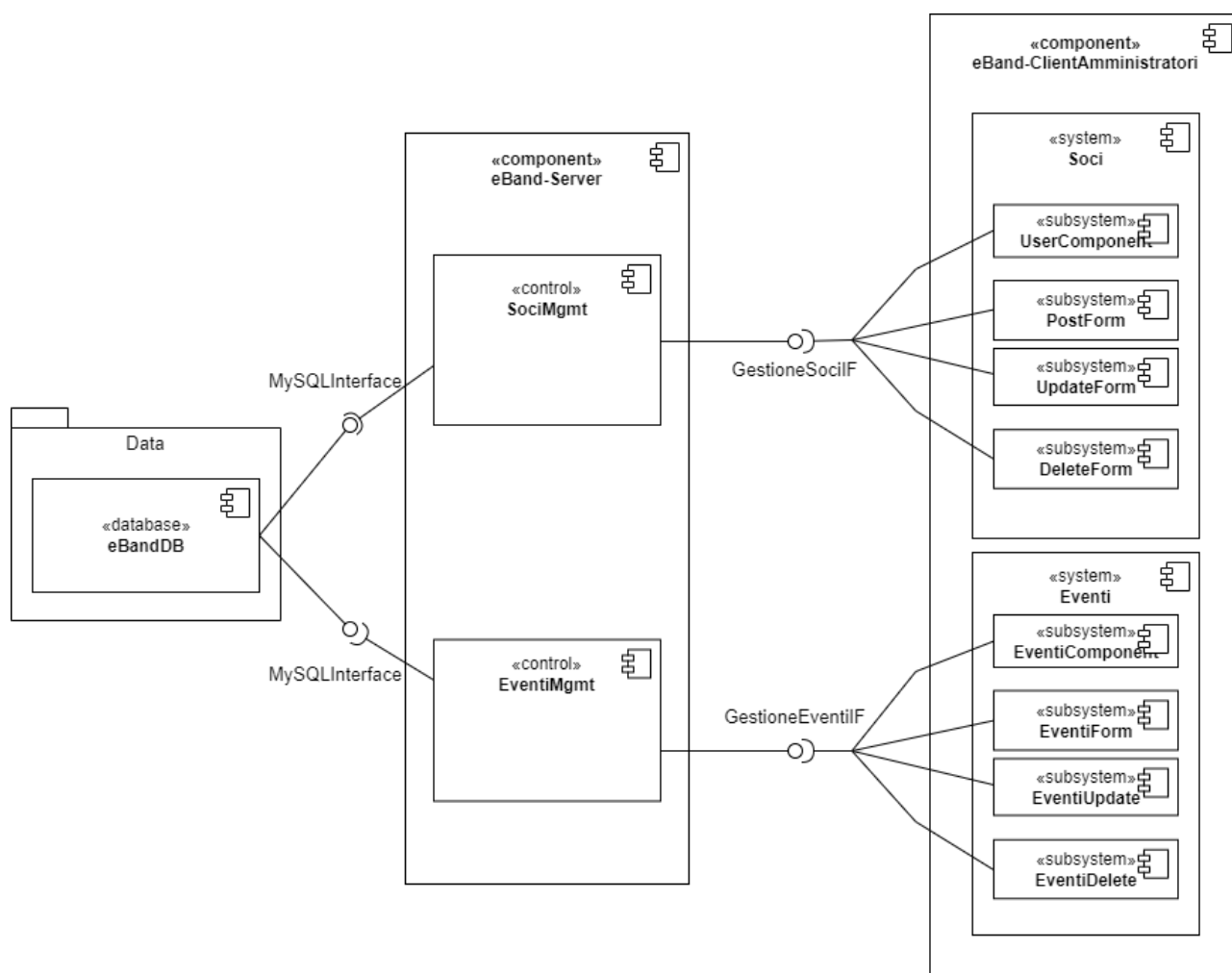
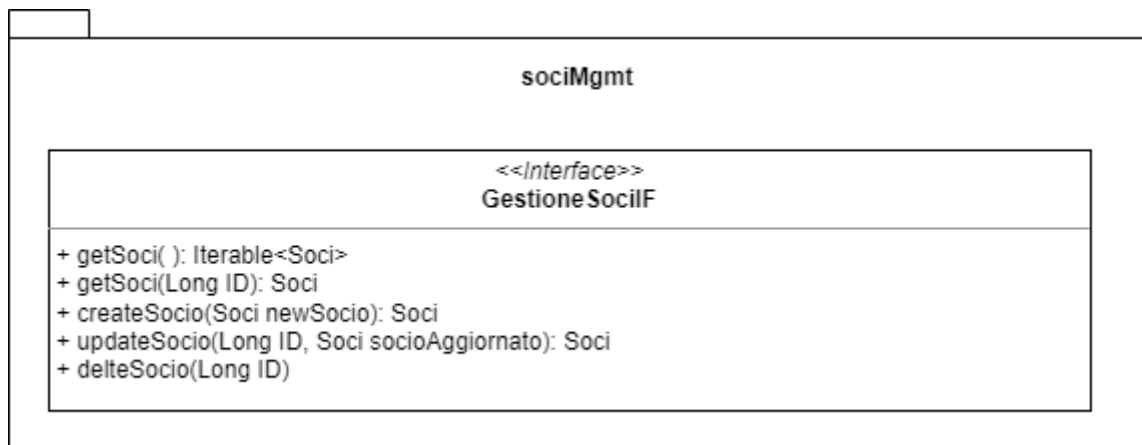


Figura 7 - Component Diagram delle parti implementate

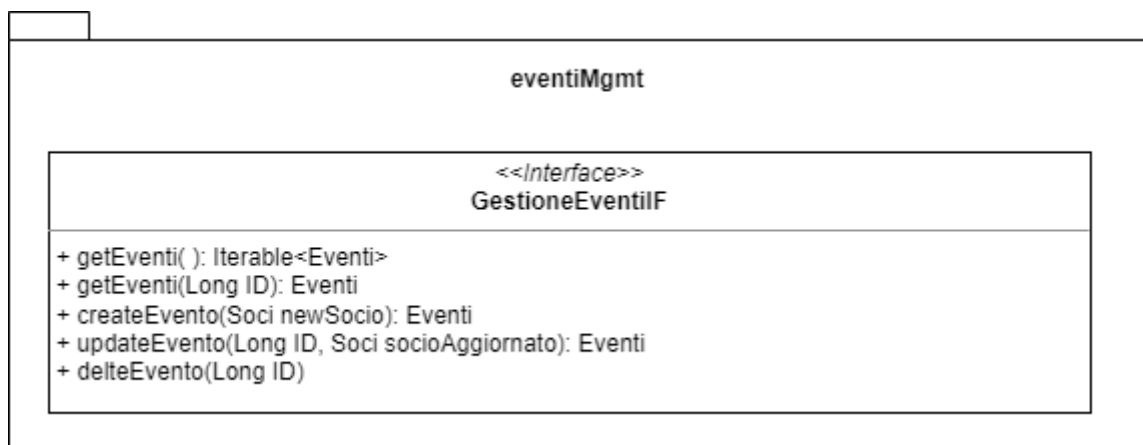
## Class Diagram dei metodi

Il class diagram seguente riferito all'interfaccia SociController implementata mette in evidenza la segnatura specifica dei metodi e i valori ritornati.



*Figura 8 - Class Diagram dei metodi interfaccia GestioneSociIF*

Il class diagram seguente invece è riferito all'interfaccia EventiController implementata che mette in evidenza la segnatura specifica dei metodi e i valori ritornati.



*Figura 9 - Class Diagram dei metodi interfaccia GestioneEventiIF*



## Tecnologie utilizzate

Questa fase implementativa ha comportato la scelta di tecnologie da utilizzare per lo sviluppo del software applicativo e per la scrittura e lettura dei dati dal database.

### MySQL Database

La scelta del database è ricaduta su MySQL, un DBMS relazionale che non richiede licenza per l'uso supportato da molti linguaggi di programmazione diffusi, tra cui Java.

### Java

Il linguaggio di programmazione scelto per lo sviluppo della parte di backend del progetto è Java. L'utilizzo di Java e delle API messe a disposizione da MySQL ha permesso di interagire con facilità con il database. Per l'implementazione delle API REST abbiamo utilizzato la tecnologia Java Maven per la sua semplicità e la disponibilità di vari framework tra cui Spring, da noi utilizzato per l'implementazione delle API REST.

### MySQL: API

Per permettere all'applicazione di interagire con il database relazionale MySQL, abbiamo specificato nel file *pom.xml* del progetto creato con framework Spring le dependencies allo scopo di rendere possibile la connessione senza l'utilizzo di librerie esterne, come illustrato di seguito.

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

*Figura 10 - Estratto del file pom.xml*

## Testing API tramite PostMan

La verifica del buon funzionamento delle API REST esposte dai vari *controller* è stata effettuata tramite il software *PostMan*. In particolare è stato testato il funzionamento di una chiamata GET esposta da *SociController* attraverso l'interfaccia *SociMngr* e di una chiamata POST esposta da *EventiController* tramite l'interfaccia *EventiMngr*.

### Test chiamata GET Soci

La chiamata GET mostrata in figura consente alla WebApp di recuperare le informazioni di un singolo socio passando il relativo id come parametro; il metodo invocato è *getSoci(Long sociID)*, cioè un overload del metodo più generico *getSoci()* utilizzato per mostrare a video nell'apposita sezione l'elenco completo di tutti i soci salvati a sistema.

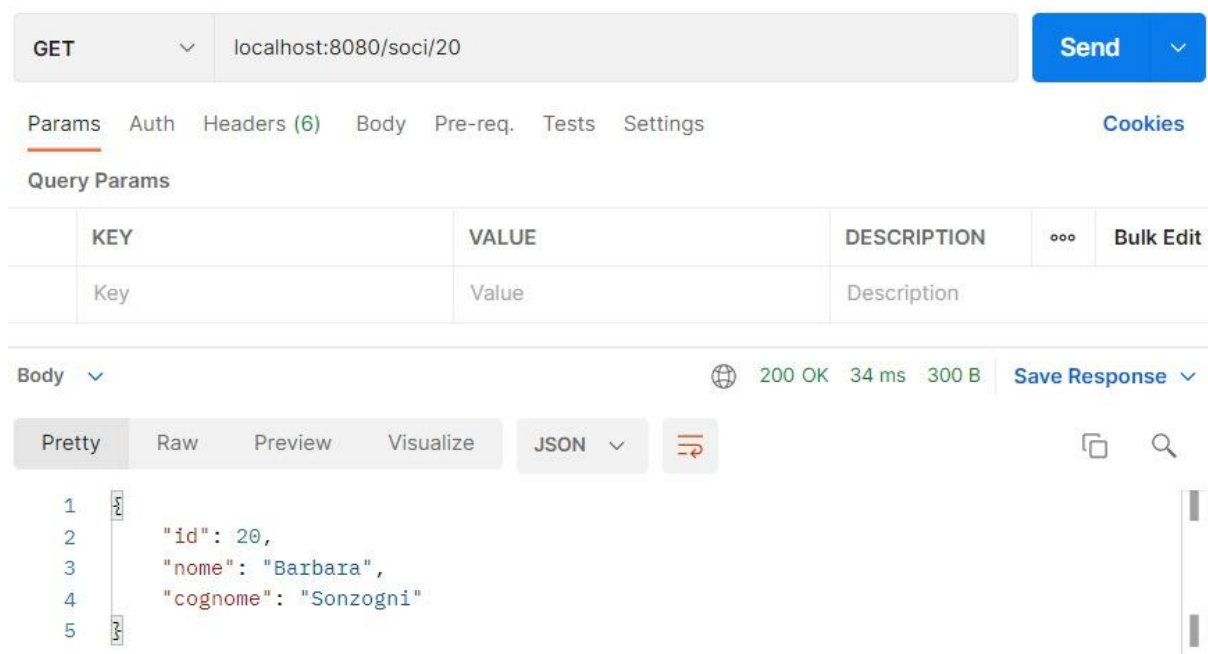


Figura 11 - chiamata GET

## Test chiamata POST Eventi

La chiamata POST mostrata in figura consente alla WebApp di salvare nel database le informazioni di un singolo evento passando i valori inseriti dall'utente nei relativi campi. Il metodo invocato è *createEvento(Eventi newEvento)*, dove *newEvento* è un parametro appartenente alla classe “*Eventi*” costruita appositamente per questo scopo.

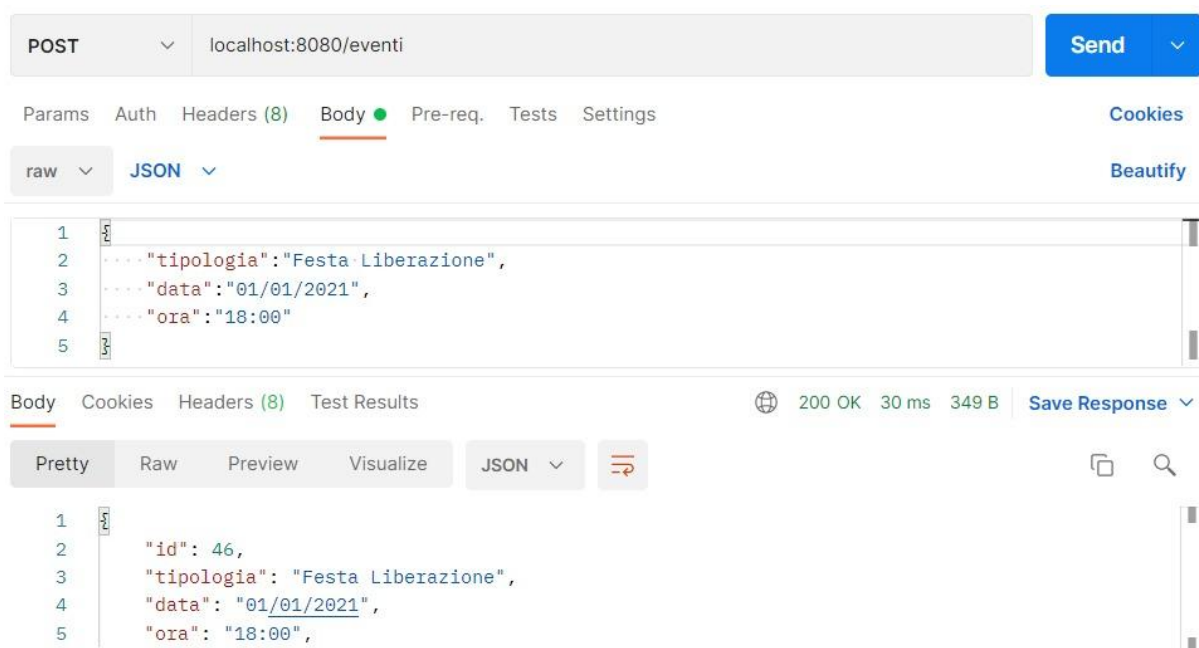


Figura 12 - chiamata POST

## Analisi Dinamica

Per effettuare l'analisi dinamica dell'applicazione è stato utilizzato JUnit: un framework di unit testing per il linguaggio di programmazione Java che ha permesso di verificare la corretta esecuzione dei casi di test previsti. Durante l'iterazione 2 sono state create le classi Soci ed Eventi con i relativi metodi get e set su ogni attributo della classe, per questo è

stata creato una classe: *EBandApplicationTestGetAndSet.java* che testa il corretto funzionamento di questi metodi.

Una seconda classe: *EBandApplicationTestAPI.java* si occupa invece di testare il corretto funzionamento delle API REST implementate nella classe *SociController* e *EventiController*.

Come si può vedere nelle due figure seguenti tutti i test previsti risultano superati.



The image displays the JUnit test results for the `EBandApplicationTestsAPI` class. The top section shows a summary: "Finished after 6,437 seconds", "Runs: 10/10", "Errors: 0", and "Failures: 0". Below this, a list of test methods is shown with their execution times: `updateEvento()` (0,410 s), `deleteEvento()` (0,180 s), `deleteSocio()` (0,025 s), `getSociTest()` (0,008 s), `updateSocio()` (0,010 s), `createSocio()` (0,023 s), `getEventID()` (0,007 s), `getEvents()` (0,007 s), `getSociID()` (0,007 s), and `createEvento()` (0,011 s). Below the test results, the source code for `EBandApplicationTestsAPI` is shown, starting with the package declaration `package eband.app;` and the import `import org.junit.jupiter.api.Test;`. The code includes annotations for `@SpringBootTest` and `@Autowired`, and defines a `SociController` instance `sc1`. It also includes a `@Test` method `getSociTest()` that asserts the result of `sc1.getSoci()` is not null, and another `@Test` method `deleteSocio()` that asserts the size of the `Soci` list after deleting a specific socio.

```
1 package eband.app;
2
3 import org.junit.jupiter.api.Test;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 @SpringBootTest
22 class EBandApplicationTestsAPI {
23
24     // Test Rest API Soci
25     SociRepository repo;
26
27     @Autowired
28     SociController sc1 = new SociController(repo);
29
30     Soci s1 = new Soci("Ilaria", "Boffelli", "11/10/1998", "Sedrin
31
32     @Test
33     public void getSociTest() {
34         assertNotNull(sc1.getSoci());
35     }
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52     @Test
53     public void deleteSocio() {
54         Iterable<Soci> i1 = sc1.getSoci();
55         long size = i1.spliterator().getExactSizeIfKnown();
56         sc1.deleteSocio((long) 76);
57         Iterable<Soci> i2 = sc1.getSoci();
58         long sizeafter = i2.spliterator().getExactSizeIfKnown();
59         System.out.println(sizeafter);
60         assertEquals(size-1, sizeafter);
61     }
62 }
```

Figura 13 - JUnit test delle chiamate API REST

Finished after 5,574 seconds

Runs: 14/14

Errors: 0

Failures: 0

▼ EBandApplicationTestsGetAndSet [Runner: JUnit 5] (0,324 s)

getResidenzaTest() (0,251 s)

getRuolo1Test() (0,007 s)

getRuolo2Test() (0,004 s)

setRuolo1Test() (0,004 s)

setRuolo2Test() (0,004 s)

setNomeTest() (0,004 s)

getNomeTest() (0,005 s)

setDataTest() (0,004 s)

getDataTest() (0,005 s)

setCognomeTest() (0,004 s)

setIscrizioneTest() (0,003 s)

setResidenzaTest() (0,004 s)

getIscrizioneTest() (0,004 s)

getCognomeTest() (0,004 s)

```
1 package eband.app;
2
3 import org.junit.jupiter.api.Test;
4
5 @SpringBootTest
6 class EBandApplicationTestsGetAndSet {
7
8     Soci s1 = new Soci("Ilaria", "Boffelli", "11/10/1998", "Sedrina",
9
10
11     @Test
12     public void getNomeTest() {
13         assertEquals("Ilaria", s1.getNome());
14     }
15
16     @Test
17     public void setNomeTest() {
18         s1.setNome("Jacopo");
19         assertEquals("Jacopo", s1.getNome());
20     }
21 }
```

Figura 14 - JUnit test dei metodi Get e Set

## ITERAZIONE 3

Durante questa iterazione il team si è concentrato principalmente sullo sviluppo della parte algoritmica. L'obiettivo era quello di implementare un algoritmo che, dato un evento programmato e l'insieme di presenze fornite dai soci, restituisce all'amministratore una possibile formazione per partecipare a quell'evento.

### **Descrizione del caso d'uso.**

#### **UC3: Gestione Organico relativo ad un certo Evento**

*Breve descrizione:* l'amministratore deve poter sapere se l'associazione ha persone a sufficienza per poter partecipare ad un determinato evento inserito a sistema.

*Attori coinvolti:* Amministratore, Sistema

*Procedimento:* L'amministratore si trova nella pagina "EventOptimizer", inserisce l'ID dell'evento di cui vuole vedere la formazione nel primo form della pagina e preme il pulsante "Calcola formazione e visualizza elenco". Il sistema esegue l'algoritmo e mostra a video il risultato del calcolo, indicando il numero di strumenti per ogni categoria. In alternativa, inserendo sempre l'id dell'evento (questa volta nel secondo form della pagina) e premendo il pulsante "Calcola formazione e visualizza elenco" il sistema esegue l'algoritmo e mostra a video il risultato del calcolo, mostrando questa volta nome, cognome e ruolo di tutti i soci selezionati.

#### UC4: Algoritmo Gestione Organico

*Breve descrizione:* l'amministratore richiede al sistema di calcolare se l'associazione è in grado di partecipare ad un determinato evento. Il sistema, in base alle presenze raccolte in precedenza, determina se per l'evento indicato il numero di persone che hanno dichiarato la loro presenza è sufficiente a soddisfare dei requisiti minimi stabiliti a priori. L'algoritmo si basa sull'assunzione che ogni socio sia in grado di suonare almeno uno strumento e al più due strumenti musicali. E' stata imposta come formazione minima la presenza di almeno 3 musicisti per ogni gruppo di strumenti principali della banda come mostrato di seguito:

Gruppi di strumenti e relativi musicisti minimi:

- |               |                  |
|---------------|------------------|
| ● Flauti      | numero minimo: 3 |
| ● Clarinetti  | numero minimo: 3 |
| ● Trombe      | numero minimo: 3 |
| ● Sax         | numero minimo: 3 |
| ● Tube        | numero minimo: 3 |
| ● Percussioni | numero minimo: 3 |
| ● Totale      | 18               |

*Attori coinvolti:* Sistema

# Algoritmo

L'algoritmo si compone di 3 fasi come illustrato di seguito.

- **Passo 1**

Nel primo passo l'algoritmo inserisce in formazione tutti coloro che sono in grado di suonare uno e un solo strumento, questi infatti sono meno "flessibili" in quanto possono essere inseriti in una e una sola categoria.

- **Passo 2**

Nel secondo passo l'algoritmo inserisce in formazioni basandosi sul ruolo principale fino al completamento del numero minimo di strumenti per ogni gruppo musicale prendendo, dove necessario per raggiungere il numero minimo, i musicisti con il loro ruolo secondario.

- **Passo 3**

Nel passo finale se si è raggiunta la formazione minima per ogni gruppo e ci sono ancora musicisti che hanno confermato la loro presenza, questi vengono inseriti in formazione sulla base del ruolo primario, altrimenti viene segnalata l'impossibilità di partecipare all'evento



## Pseudocodice

```
1 algoritmo CalcolaFormazione (array ListaPresenti[1..n] di Soci) --> array di ResAlgoritmo
2   array risultato[1..m] di ResAlgoritmo
3
4   //Passo 1
5   foreach socio in ListaPresenti do
6     if (ruolo2 di socio = NULL) then
7       risultato <-- risultato U {socio}
8       ListaPresenti <-- ListaPresenti - socio
9     endif
10  endfor
11
12  //Passo 2
13  for x=0 to 1 do
14    foreach socio in ListaPresenti do
15      ruolo //stringa contenente il ruolo selezionato
16      if (x=0) then
17        ruolo <-- ruolo1 di socio
18      else
19        ruolo <-- ruolo2 di socio
20      endif
21
22      if (vincolo ruolo raggiunto) then:
23        continue
24      endif
25
26      if (socio ∉ risultato) then:
27        risultato <-- risultato U {socio}
28        ListaPresenti <-- ListaPresenti - socio
29      endif
30
31    endfor
32  endfor
33
34  //Passo 3
35  if( card(risultato) < vincolo) then
36    return "Evento non fattibile"
37  else
38    foreach socio in ListaPresenti do
39      ruolo <-- ruolo1 di socio
40      if (socio ∉ risultato)
41        risultato <-- risultato U {socio}
42      endif
43    endfor
44  endif
45  return risultato
```

*Figura 15 - Pseudocodice dell'algoritmo*

## Costo Computazionale

Analizziamo la complessità dell'algoritmo ponendoci nel caso peggiore e supponendo di aggregare tutte le operazioni elementari al di sotto di un'unica costante  $C$ .

In corrispondenza del passo 1, nel caso peggiore (ma anche in quello migliore), l'algoritmo dovrà scansionare tutto l'array in input. Il suo costo sarà quindi  $\Theta(n)$ , con  $n$  la dimensione dell'array.

In corrispondenza del passo 2, nel caso peggiore, l'algoritmo avrà costo pari a  $\Theta(n^2)$ . Per spiegarlo bisogna ricordare ciò che avviene al passo 1: in questa fase l'algoritmo, man mano che inserisce un socio nell'array di output, lo elimina dall'array di input, mantenendo perciò la somma delle dimensioni dei due array sempre pari a  $n$ . All'inizio del passo 2 quindi viene scansionato il vettore di input (che adesso avrà dimensione pari a  $n - a$ , con  $0 \leq a \leq n$ ) e successivamente, in corrispondenza dell'ultimo if del passo 2, il controllo che verifica che l' $i$ -esimo socio non appartenga all'array di output viene eseguito scorrendo il vettore finale, di dimensione pari ad  $a$ .

Il costo quindi è così calcolato (trascurando l'eventuale costante  $C$ ):

$$\Theta(n - a) * \Theta(a)$$

Nel caso in cui  $a = 0$ , il costo è pari a  $\Theta(n)$

Nel caso in cui  $a = n$ , il costo è pari a  $\Theta(n)$

Nel caso (peggiore) in cui  $a = n/2$ , il costo è pari a

$$\Theta(n/2) * \Theta(n/2) = \Theta(n^2/4) = \Theta(n^2)$$

Infine, per quanto riguarda il passo 3, il caso peggiore è dato dal ramo else che segue un ragionamento simile a quello illustrato al passo 2. Il relativo costo pertanto è pari a  $\Theta(n^2)$ .

## UML - Component Diagram delle componenti implementate

Le componenti che implementano le funzionalità scelte per l'implementazione nell'iterazione 3 sono descritte nel *component diagram* seguente che evidenzia le interazioni tra i componenti dal punto di vista della funzionalità.

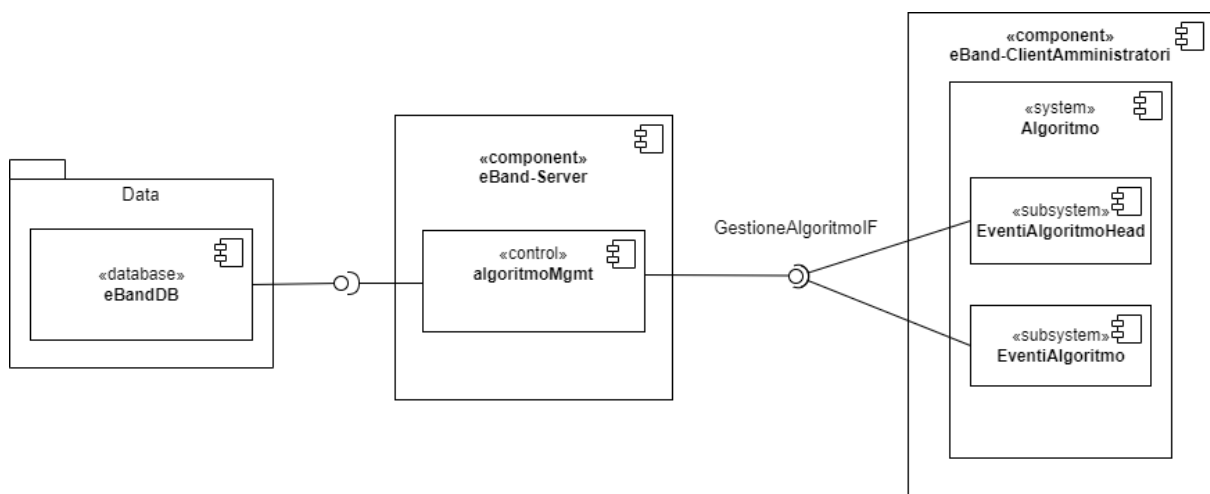


Figura 16 - Component Diagram dell'oggetto implementato

## Class Diagram dei metodi

Il class diagram seguente riferito all'interfaccia AlgoritmoController implementata mette in evidenza la segnatura specifica dei metodi e i valori ritornati.

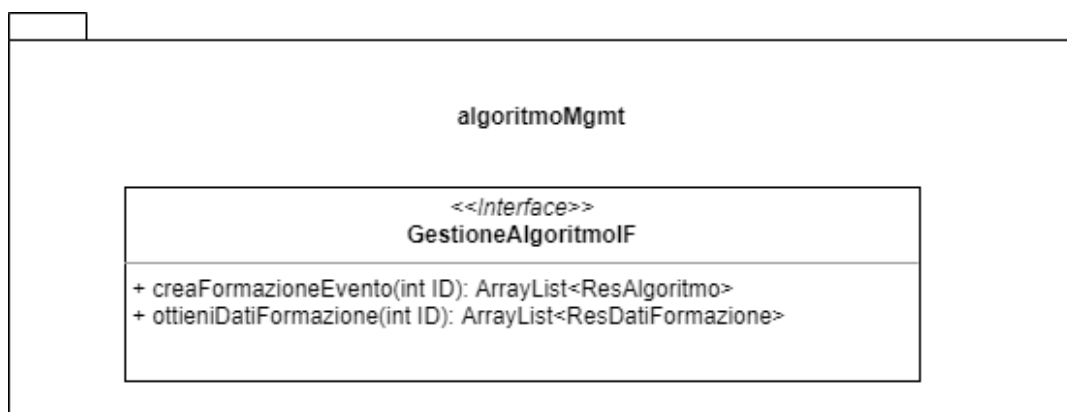


Figura 17 - Class Diagram dei metodi interfaccia AlgoritmoController

## UML Sequence Diagram

Di seguito viene presentato il sequence diagram sviluppato che mette in evidenza l'interazione tra il client, il server e il Database in seguito alla richiesta di esecuzione del calcolo della formazione di un evento specificato tramite ID da parte dell'utente.

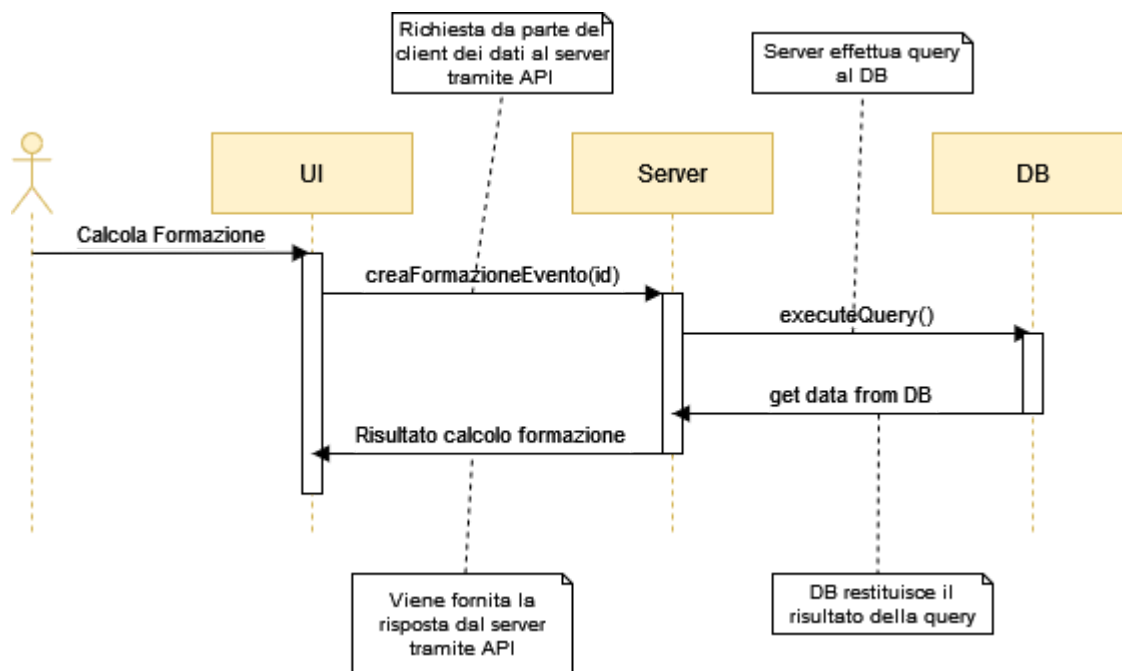


Figura 18 - Sequence Diagram

Si può notare che la richiesta tramite API del client viene implementata nel server chiamando il metodo 'creaFormazioneEvento' dell'interfaccia GestioneAlgoritmoIF. Chiamando il metodo viene eseguita la connessione al database e salvato il risultato ottenuto dall'esecuzione della query. Nel server viene eseguito l'algoritmo di calcolo della formazione che restituisce un oggetto JSON al client.

## Test chiamata GET Algoritmo

La chiamata GET mostrata in figura consente all'interfaccia grafica di mostrare a video nell'apposita sezione il risultato dell'elaborazione dell'algoritmo. In particolare per l'ottenimento di questa informazione viene invocato il metodo *creaFormazioneEvento()*.

The screenshot displays a REST client interface with the following components:

- Request Bar:** Method: GET, URL: localhost:8080/esegui?id=26, Send button.
- Params Tab:** Query Params table with one entry: id = 26.
- Response Tab:** Status: 200 OK, 361 ms, 1.81 KB. Response body is a JSON array.

KEY	VALUE	DESCRIPTION
id	26	

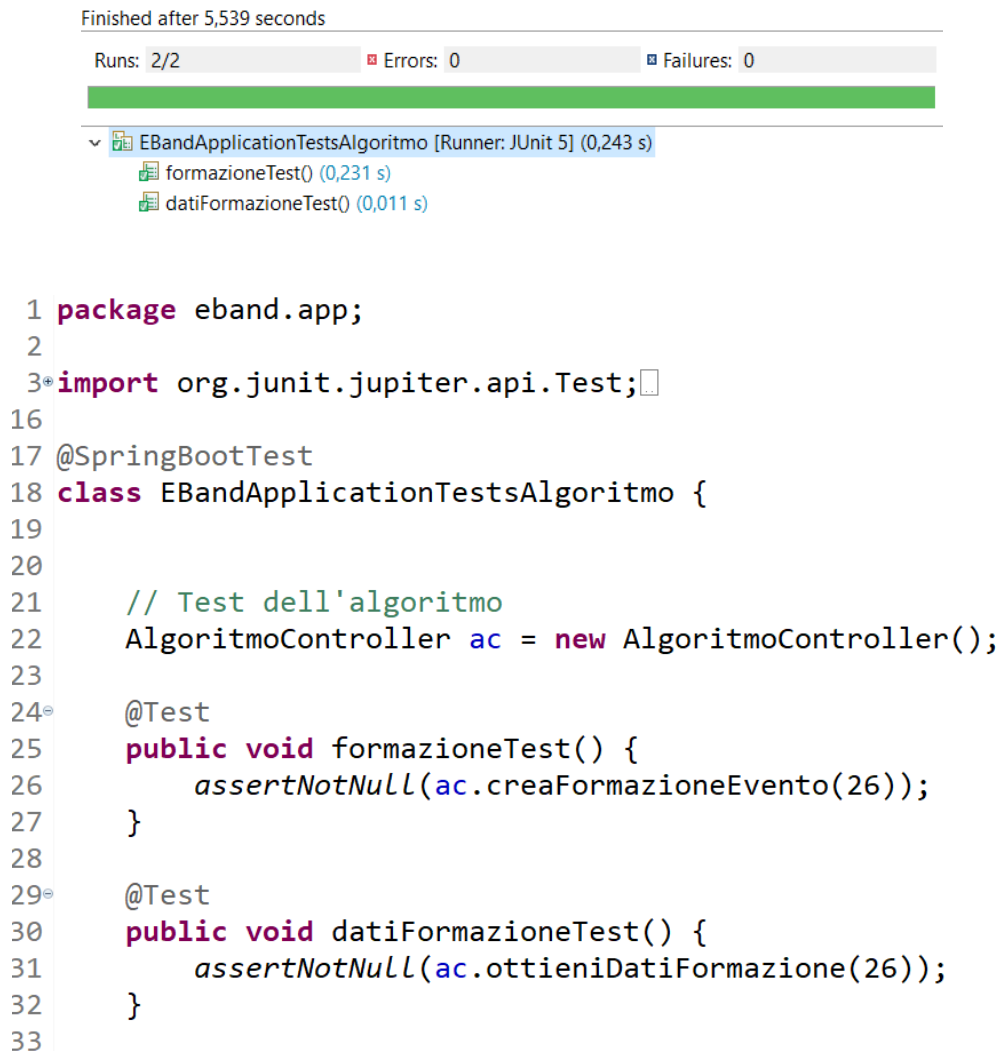
```
{
  "id": 27,
  "cognome": "Cavalleri",
  "nome": "Yari",
  "ruolo": "tuba"
},
{
  "id": 40,
  "cognome": "Capelli",
  "nome": "Enrico",
  "ruolo": "flauto"
}
```

Figura 19 - Test chiamata GET con l'utilizzo di POSTMAN

## Analisi dinamica

Come nell'iterazione precedente ci siamo concentrati sulla verifica con JUnit Test dei metodi implementati per lo sviluppo dell'algoritmo.

Le seguenti immagini mostrano come i metodi implementati hanno superato positivamente i test creati.



The image shows a screenshot of a JUnit test runner interface. At the top, it says "Finished after 5,539 seconds". Below that, a summary bar shows "Runs: 2/2", "Errors: 0", and "Failures: 0". A green progress bar is also visible. The test results list shows a tree view with "EBandApplicationTestsAlgoritmo [Runner: JUnit 5] (0,243 s)" expanded, showing two sub-items: "formazioneTest() (0,231 s)" and "datiFormazioneTest() (0,011 s)". Below the test results, the source code for the test class is displayed with line numbers 1 through 33. The code is in Java and uses JUnit 5 annotations like @SpringBootTest and @Test. It defines a class EBandApplicationTestsAlgoritmo with two test methods: formazioneTest() and datiFormazioneTest(), both using assertNotNull() to verify the results of the algorithm's methods.

```
1 package eband.app;
2
3 import org.junit.jupiter.api.Test;
4
16
17 @SpringBootTest
18 class EBandApplicationTestsAlgoritmo {
19
20     // Test dell'algoritmo
21     AlgoritmoController ac = new AlgoritmoController();
22
23     @Test
24     public void formazioneTest() {
25         assertNotNull(ac.creaFormazioneEvento(26));
26     }
27
28     @Test
29     public void datiFormazioneTest() {
30         assertNotNull(ac.ottieniDatiFormazione(26));
31     }
32 }
33
```

Figura 20 - JUnit test dei metodi che richiamano l'algoritmo

## Analisi statica del codice (CodeMR)

Al termine dell'iterazione 3 è stata effettuata un'analisi statica del codice mediante il tool CodeMR. Questo strumento permette di fare un'analisi statica del codice del progetto nella sua interezza, esplorando moduli e pacchetti presenti evidenziando quali e quante dipendente possiedono. Riportiamo alcuni grafici e viste generate che mostrano alcune caratteristiche e metriche del codice al termine dell'iterazione 3.

### Distance: legame tra Abstractness e Instability

La seguente metrica è stata calcolata relativamente al package algoritmoMgmt, il grafico mostra lo scostamento del valore dal caso ideale, minore è la distanza del punto identificato rispetto alla linea maggiore è la qualità del software.

Il grafico visualizza il valore delle metriche prese in considerazione, il tool ha autonomamente eseguito il calcolo della metrica Normalized Distance utilizzando la seguente formula:  $ND = | \text{Abstractness} + \text{Instability} - 1 |$

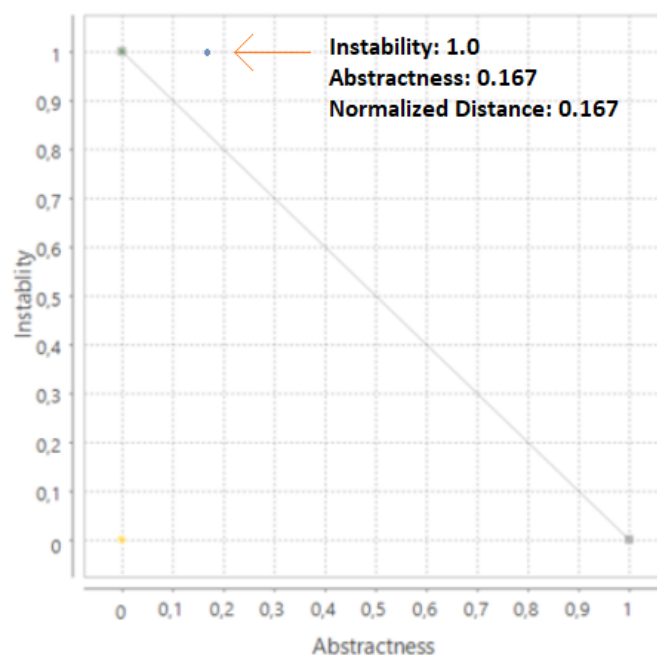


Figura 21 - Abstractness and Instability

## Treemap View

La seguente visualizzazione mostra le dipendenze tra package con un approccio visivo legato ai colori e alla posizione dei blocchi nell'immagine. Il codice rappresentato da uno dei blocchi, dipende dal codice rappresentato dal blocco sottostante e ha una dipendenza bidirezionale dai blocchi collocati sul suo stesso livello. Nell'interfaccia HTML generata dal tool è possibile interagire con il grafico e visualizzare nel dettaglio ciò che accade all'interno di un singolo package.



*Figura 22 - treemap del progetto eBand Server*

## Grafo strutturale

CodeMR permette anche la generazione di grafi per la visualizzazione della struttura del progetto Spring in Java che costituisce il lato server (backend) dell'applicazione software.



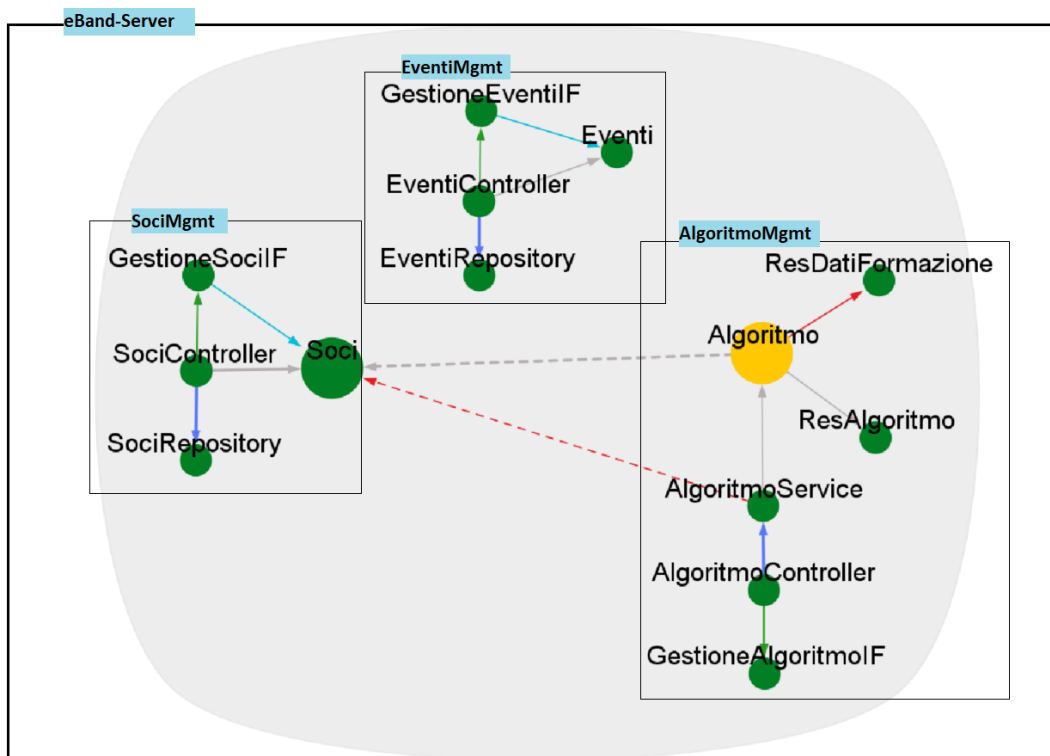


Figura 23 - grafo strutturale del progetto eBand Server

## Project Outline

Oltre ai grafici e alle varie view lo strumento di analisi statica fornisce anche uno schema riassuntivo del progetto che rappresenta i valori di molte metriche relative ad ogni classe del codice. Nella figura seguente riportiamo i valori di alcune delle metriche più importanti calcolate nel seguente ordine:

- Quality Attributes
- Line Of Code
- Coupling
- Complexity
- Size
- Lack of Cohesion

Element	Quality Attributes			LOC	Coupling	Complexity	Size	Lack of Cohesion
▼ ▲ eBand-Server								
▼ eband.app	●	■	■	4	low	low	low	low
> EBandApplication	●	■	■	4	low	low	low	low
▼ eband.app.algoritmoMgmt	●	■	■	353	low	low	low-medium	low
> Algoritmo	●	■	■	253	low	medium-high	low-medium	low
> AlgoritmoController	●	■	■	12	low	low	low	low
> AlgoritmoService	●	■	■	36	low	low	low	low
> GestioneAlgoritmoIF	●	■	■	3	low	low	low	low
> ResAlgoritmo	●	■	■	18	low	low	low	low
> ResDatiFormazione	●	■	■	31	low	low	low	low
▼ eband.app.eventiMgmt	●	■	■	60	low	low	low	low
> Eventi	●	■	■	28	low	low	low	low
> EventiController	●	■	■	25	low	low	low	low
> EventiRepository	●	■	■	1	low	low	low	low
> GestioneEventiIF	●	■	■	6	low	low	low	low
▼ eband.app.sociMgmt	●	■	■	97	low	low	low	low
> GestioneSociIF	●	■	■	6	low	low	low	low
> Soci	●	■	■	61	low	low	low-medium	low
> SociController	●	■	■	29	low	low	low	low
> SociRepository	●	■	■	1	low	low	low	low

Figura 24 - Project Outline