Object Design Document

MyBomber

Riferimento	2021_SOW_C11_MyBomber_Mauro 2021_RAD_C11_MyBomber_Mauro 2021_SDD_C11_MyBomber_Mauro 2021_TCS_C11_MyBomber_Mauro
Versione	2.0
Data	24/01/2022
Destinatario	Professori F. Ferrucci e F. Palomba
Presentato da	Bonavita Vincenzo, D'Antuono Domenico, Palladino Antonio Salvatore, Viglione Maddalena.
Approvato da	Gaetano Mauro

Revision History

Data	Versione	Descrizione	Autori
18/11/2021	0.1	Prima stesura fino a capitolo 1.1	MV
18/11/2021	0.2	Aggiunta paragrafo 1.2	VB
19/11/2021	0.3	Aggiunta capitoli 1.3 a 1.3.4	AP
27/11/2021	0.4	Aggiunta paragrafo 1.4	MV
18/11/2021	0.5	Aggiunta capitoli 2 a 2.3	VB, MV
20/11/2021	0.6	Aggiunta capitolo 3	AP
19/11/2021	0.7	Aggiunta capitoli 4 e 4.1	VB, MV
27/11/2021	0.8	Aggiunta paragrafo 4.2	DD, MV
27/11/2021	0.9	Aggiunta paragrafi 4.4 e 4.4.1	DD
27/11/2021	0.10	Aggiunta paragrafo 4.4.2	VB
07/11/2021	1.0	Revisione generale	AP, MV
19/01/2022	2.0	Revisione accurata	DD,MV

Team members

Nome Cognome	Ruolo nel progetto	Acronimo	Informazione di contatto
Gaetano Mauro	Project Manager	GM	g.mauro14@studenti.unisa.it
Vincenzo Bonavita	Team Member	VB	v.bonavita@studenti.unisa.it
Domenico D'Antuono	Team Member	DD	d.dantuono7@studenti.unisa.it
Antonio Salvatore Palladino	Team Member	AP	a.palladino48@studenti.unisa.it
Maddalena Viglione	Team Member	MV	m.viglione@studenti.unisa.it

Sommario

1. Introduzione

MyBomber si propone di semplificare le interazioni tra un giocatore di calcio con altri giocatori e con il gestore di una struttura sportiva, al fine di facilitare i processi di prenotazione per eventi, creazione di eventi o partecipazione agli eventi stessi.
Il documento che segue illustra diversi dettagli legati alla fase implementativa del sistema MyBomber. In particolare, andremo a descrivere i trade-off di progettazione definiti dagli sviluppatori, le linee guida da seguire per le interfacce dei sottosistemi, la decomposizione dei sottosistemi in pacchetti e classi, e, infine, la specifica delle interfacce delle classi.

1.1 Object Design Trade-offs

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.

Manutenibilità vs Performance:
Il sistema sarà progettato in modo tale da permettere, in futuro e quando se ne necessiterà, una semplice manutenzione. Per ottenere ciò si dovrà prediligere una scrittura del codice volta alla modularità delle sue parti.

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.

Funzionalità vs Usabilità:
Verrà realizzata un'interfaccia grafica chiara e concisa, usando form e pulsanti predefiniti che hanno lo scopo di rendere semplice l'utilizzo del sistema da parte dell'utente finale, delegando a pagine specifiche le funzionalità più avanzate che minano l'uso intuitivo del software.

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.

1.2 Componenti off-the-shelf

Per lo sviluppo del sistema è previsto l'uso di diversi componenti off-the-shelf, ovvero componenti software messi a disposizione dal mercato che offrono pacchetti di soluzioni che possono essere utili a risolvere degli specifici problemi. Le componenti previste per la realizzazione del sistema sono le seguenti:

- *Bootstrap*, un toolkit open source utilizzato per lo sviluppo di progetti responsive sul web. Il suo utilizzo è previsto insieme a quello di HTML, CSS e JavaScript per realizzare la struttura base della grafica.

- *jQuery.js*, una libreria JavaScript per applicazioni web il cui obiettivo è quello di semplificare la selezione, la manipolazione, la gestione degli eventi e l'animazione di elementi DOM in pagine HTML. Il suo utilizzo ha lo scopo di ridurre la complessità del codice JavaScript durante l'implementazione.

- *Selenium*, una suite di tool utilizzati per automatizzare i test del sistema eseguendoli su un browser web. Esso viene utilizzato per eseguire le attività di testing di sistema.

- *JUnit*, un framework di programmazione Java che viene utilizzato per implementare i test di unita.

- *Mockito*, un tool utilizzato per i mock degli oggetti durante il testing di unità.

1.3 Linee guida per la documentazione delle interfacce

1.3.1 Package Model e Control

Il progetto verrà realizzato con l'IDE di sviluppo e sarà strutturato nel seguente modo:

- Nel progetto vi sono due package (Model, Control), i quali contengono i rispettivi subpackage con le rispettive classi.
- Le classi sviluppate rispetteranno il Checkstyle di Google, adottando quindi il seguente schema (per altre informazioni: https://checkstyle.sourceforge.io/google_style.html):
    1. Clausole import
    2. Dichiarazione della classe
    3. Variabili di istanza
    4. Costruttore
    5. Metodo
- Il nome di una classe deve rispettare il seguente formato: NomeClasse.
- Il nome di un metodo o di una variabile di istanza deve rispettare il seguente formato: nomeDellaVariabile, nomeDelMetodo.

- I nomi utilizzati per identificare classi, attributi e metodi devono risultare quanto più possibile significativi per quanto riguarda il loro scopo.
- Una parentesi graffa chiusa è preceduta e seguita da un carattere di new line.
- Un costrutto di tipo if che prevede una sola istruzione può evitare l'uso delle parentesi graffe.
- I commenti possono essere utilizzati soltanto nei casi ritenuti opportuni. Nel caso in cui un commento si estenda per una sola riga, esso presenta il formato: // commento; se, invece, un commento si estende su più righe diverse, esso presenta il formato /* commento */.
- Ogni classe e ogni metodo devono essere corredate da commenti che rispettano lo standard utilizzato da Javadoc per la produzione di documentazione in formato HTML.
- Il package Model contiene tutte le classi che fanno riferimento ad entità persistenti (Bean e DAO).
- Il package Control contiene tutte le classi Servlet che si occupano della logica di business del sistema e agisce da interlocutore tra le classi contenute nel package Model e contiene la classe DriverManagerConnectionPool che si occupa della connessione del sistema al Database
- Dov'è possibile, in modo particolare per i Bean contenuti nel package Model, le classi sono corredate da opportuni metodi GETTER, SETTER e, eventualmente, sovrascrivono il metodo toString().

1.3.2 JavaScript

Nel progetto vengono utilizzate pagine JavaScript, che fanno da intermediari tra pagine JSP e Servlet.

- Gli Script in JavaScript devono rispettare le seguenti convenzioni:
  1. Gli script che svolgono funzioni distinte dal vero rendering della pagina dovrebbero essere collocati in file dedicati.
  2. Le funzioni JavaScript devono essere documentate in modo analogo ai metodi Java.

1.3.3 Interfaccia Grafica

Le interfacce del progetto vengono implementate attraverso i file JSP, HTML e CSS.

- Le pagine JSP devono, quando eseguite, produrre un documento. Il codice Java delle pagine deve aderire alle convenzioni per la codifica in Java, con le seguenti puntualizzazioni:
  1. Il tag di apertura () si trova all'inizio di una riga;
  2. Il tag di chiusura (%>) si trova all'inizio di una riga;
  3. È possibile evitare le due regole precedenti, se il corpo del codice Java consiste in una singola istruzione ().
- Le pagine HTML, sia in forma statica che dinamica, devono essere conformi allo standard HTML5. Inoltre, il codice HTML statico deve utilizzare l'indentazione, per facilitare la lettura, secondo le seguenti regole:
  1. Un'indentazione consiste in una tabulazione;
  2. Ogni tag deve avere un'indentazione maggiore del tag che lo contiene;
  3. Ogni tag di chiusura deve avere lo stesso livello di indentazione del corrispondente tag di apertura;

4. I tag di commento devono seguire le stesse regole che si applicano ai tag normali.

- Fogli di stile CSS devono seguire le seguenti convenzioni:
  1. Tutti gli stili non inline devono essere collocati in fogli di stile separati.
  2. Ogni foglio di stile deve essere iniziato da un commento analogo a quello presente nei file Java.
  3. Ogni regola CSS deve essere formattata come segue:
     - I selettori della regola si trovano a livello 0 di indentazione, uno per riga;
     - L'ultimo selettore della regola è seguito da parentesi graffa aperta ({);
     - Le proprietà che costituiscono la regola sono listate una per riga e sono indentate rispetto ai selettori;
     - La regola è terminata da una parentesi graffa chiusa (}), collocata da sola su una riga.

1.3.4 Database SQL

- I nomi delle tabelle del Database SQL devono seguire le seguenti regole:
  1. Devono essere costituiti da sole lettere minuscole;
  2. Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.
- I nomi dei campi devono seguire le seguenti regole:
  1. Devono essere costituiti da sole lettere minuscolo;
  2. Se il nome è costituito da più parole, è previsto l'uso di underscore (_);
  3. Il nome deve essere un sostantivo singolare tratto dal dominio del problema ed esplicativo del contenuto.

1.4 Glossario

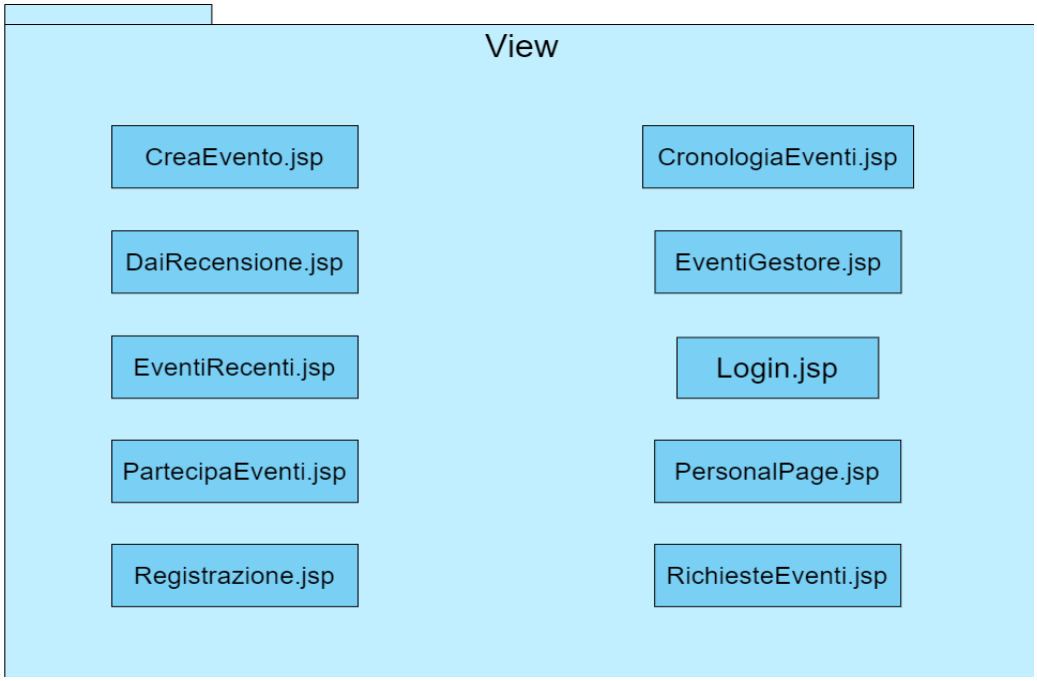
Sigla/Termine	Definizione
Package	Raggruppamento di classi, interfacce o file correlati.
DAO	Data Access Object, implementazione dell'omonimo pattern architetturale che si occupa di fornire un accesso in modo astratto ai dati persistenti.
Controller	Classe che si occupa di gestire le richieste effettuate dal client.
Model	Parte del design architetturale MVC che fornisce al sistema i metodi per accedere ai dati utili al sistema.
MVC	Model-View-Controller: design architetturale che permette di separare la logica di presentazione dalla logica di business alla base del sistema.
Design pattern	Template di soluzioni a problemi ricorrenti impiegati per ottenere riuso e flessibilità.
Adapter	È un pattern strutturale che può essere basato sia su classi che su oggetti il cui fine è fornire una soluzione astratta al problema

	dell'interoperabilità tra interfacce differenti.
Singleton	È un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga struttura una solo istanza e di fornire un punto di accesso globale a tale istanza.
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.

2. Packages

2.1 View

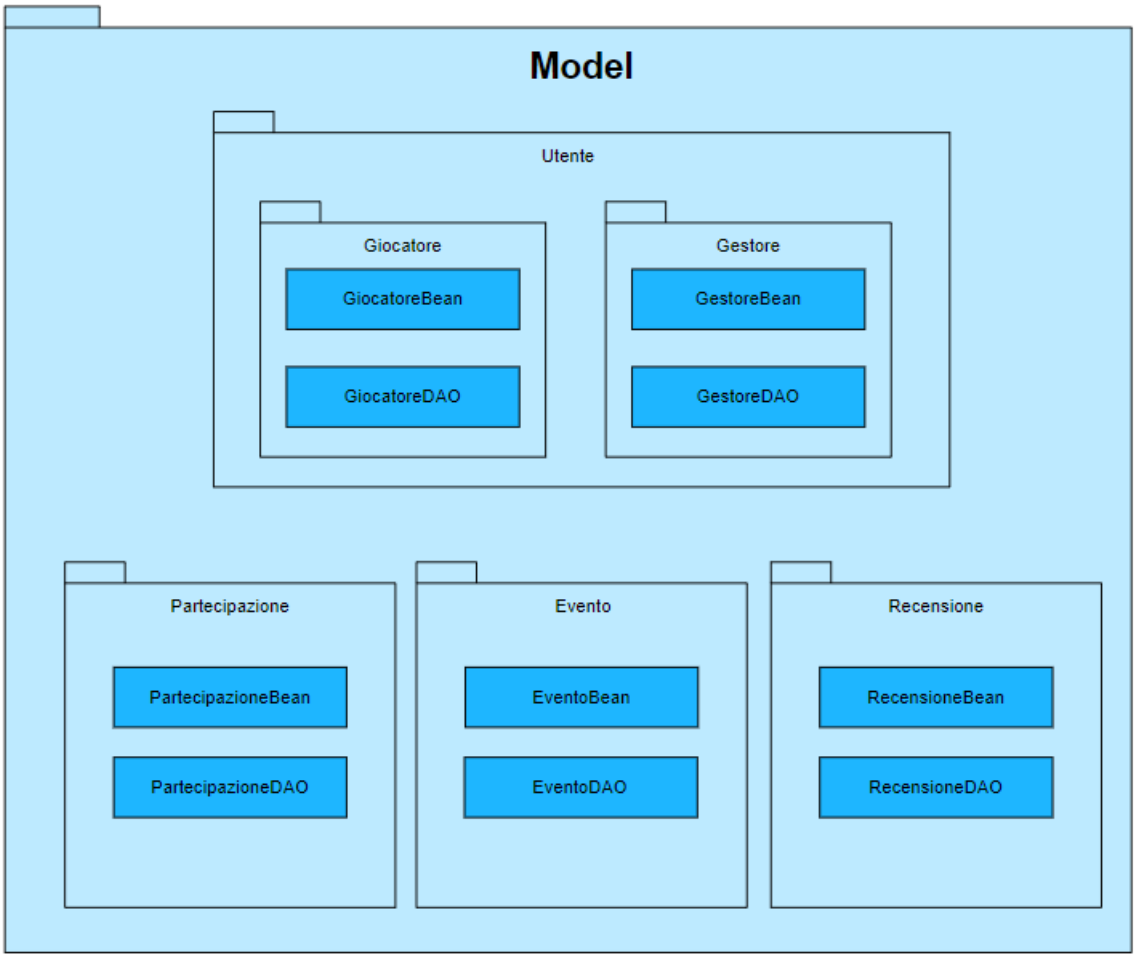
Il package View contiene le JSP per la visualizzazione delle pagine, la classe *Homepage.jsp* viene utilizzata per mostrare la prima interfaccia che è possibile visualizzare accedendo al server: grazie ad essa sarà possibile registrarsi o accedere alla propria pagina personale tramite le *Login.jsp*. Le pagine *RegistrazioneGiocatore.jsp* e *RegistrazioneGestore.jsp* sono rispettivamente per la registrazione di un nuovo giocatore e un nuovo gestore. La *PersonalPage.jsp* mostra le informazioni dell'utente loggato. Per la creazione dell'evento vi è la pagina *CreaEvento.jsp*. Inoltre, sono presenti le pagine *CronologiaEventi.jsp*, *VisualizzaEventi.jsp* e *EventiRecenti.jsp* pensate per la visualizzazione degli eventi passati, la visione degli eventi ai quali si può partecipare(per lato giocatore) o semplicemente controllare (per lato gestore) e per visualizzare gli eventi recenti a cui un giocatore ha partecipato. *GestioneEvento.jsp* è la page creata per il gestore dove il suddetto può accettare o declinare una richiesta di evento stesso, mentre *EventiAttiviStruttura.jsp* è stata pensata per poter accedere in modo rapido (sempre per il gestore) agli eventi che si stanno tenendo in struttura. *DaiRecensione.jsp* e *VisualizzaRecensione.jsp* sono state create per appunto gestire la recensione e la prima permette appunto di recensire un utente, mentre la seconda permette al giocatore loggato di visualizzare una recensione a lui data.



2.2 Model

Il package Model contiene la classe DriverManagerConnectionPool che consente la connessione al database e la classe Utility dove sono presenti metodi generici necessari per il corretto funzionamento del server. Inoltre, nel package principale Model, sono presenti altri 3 package: Utente, Evento e Recensione. Giocatore e Gestore sono estensioni di Utente.

- Il **package Utente** si suddivide in due packages:
  - Il **package Giocatore** contiene la classe GiocatoreBean che rappresenta un Giocatore, la classe GiocatoreDAO che rappresenta l'interfaccia che contiene i metodi utili all'interazione con il database per ciò che riguarda la figura del Giocatore.
  - Il **package Gestore** contiene la classe GestoreBean che rappresenta un Gestore, la classe GestoreDAO che rappresenta l'interfaccia che contiene i metodi utili all'interazione con il database per ciò che riguarda la figura del Gestore.
- Il **package Evento** contiene la classe EventoBean che rappresenta un Evento, la classe EventoDAO che contiene i metodi utili all'interazione con il database per ciò che riguarda l'entità Evento.
- Il **package Recensione** contiene la classe RecensioneBean che rappresenta una Recensione la classe RecensioneDAO che contiene i metodi utili all'interazione con il database per ciò che riguarda l'entità Recensione.
- Il **package Partecipazione** contiene la classe PartecipazioneBean che rappresenta la partecipazione ad un evento, la classe PartecipazioneDAO contiene i metodi utili all'iterazione con il database per quanto riguarda la partecipazione.



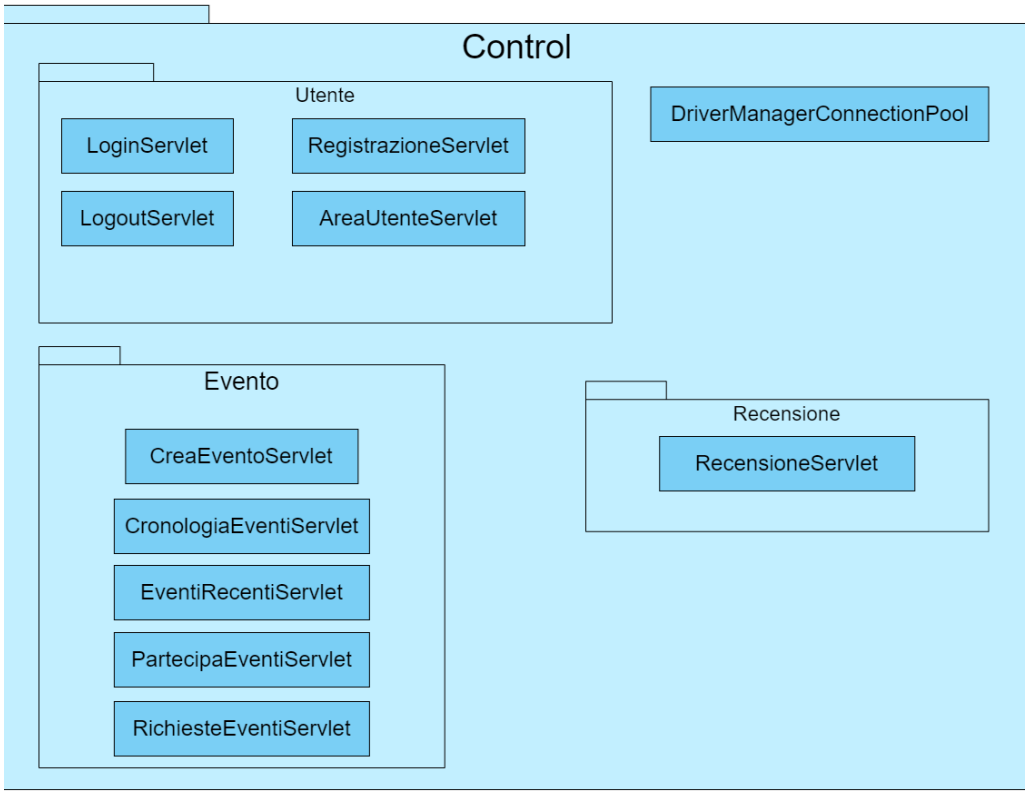
2.3 Control

Il package Control riceve, tramite il pacchetto View, i comandi dall'utente. È formato a sua volta da tre package: Utente, Evento e Recensione. Giocatore e Gestore sono estensioni di Utente.
Tutti i package gestiranno l'attività del server consentendo all'utente di interagire con il sistema.

- Il **package Utente** contiene le servlet:
  - LoginServlet, si occupa dell'accesso alla piattaforma;
  - LogoutServlet, riguarda la disconnessione dal proprio account ;
  - AreaUtenteServlet, contiene il profilo dell'utente;
  - RegistrazioneServlet, permette di registrarsi alla piattaforma creando il proprio account.
- Il **package Evento** contiene le servlet:
  - CreaEventoServlet (lato giocatore), per la creazione di eventi;
  - CronologiaEventiServlet (lato gestore), per la visualizzazione degli eventi sia attivi che completati;
  - EventiRecentiServlet (lato giocatore), per la visualizzazione degli eventi attivi e completati del singolo utente;
  - PartecipaEventiServlet (lato giocatore), per poter partecipare a degli eventi attivi sulla piattaforma;
  - RichiesteEventiServlet (latore gestore), per poter accettare o rifiutare le richieste di creazioni di nuovi eventi, fatte sulla struttura di quel determinato gestore.
- Il **package Recensione** contiene

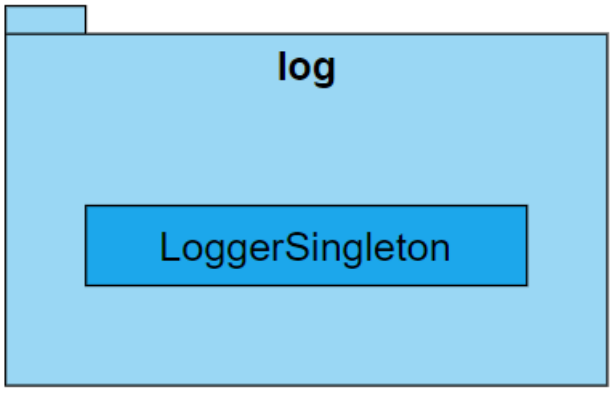
- RecensioneServlet (lato giocatore) per la creazione di una recensione;

- DriverManagerConnectionPool, per la connessione al database.



2.4 Log

Il package log contiene il file LoggerSingleton.java che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza e di fornire un punto di accesso globale a tale istanza. In questo modo abbiamo la possibilità di stampare i valore delle variabili o dei messaggi all'interno di un file, tramite il LoggerSingleton. Nel nostro caso viene previsto un unico costruttore privato e la classe fornisce inoltre un metodo getInstance() statico che restituisce l'istanza della classe, creandola preventivamente alla prima chiamata del metodo.



3. Class Interfaces

Nome Classe	LoginServlet
Descrizione	Questa classe crea una sessione per l'utente registrato.
Pre:	**Context** LoginServlet::doPost(request,response); **Pre:** request.getParameter("e-mail") != NULL && request.getParameter("password") != NULL
Post:	**Context** LoginServlet::doPost(request,response) **Post:** request.getParameter("giocatore") != NULL | | request.getParameter("gestore") != NULL;
Invarianti	

Nome Classe	LogoutServlet
Descrizione	Questa classe invalida la sessione dell'utente che ha effettuato il login.
Pre:	**Context** LogoutServlet::doGet(request,response); **Pre:**
Post:	**Context** LogoutServlet::doGet(request,response) **Post:**
Invarianti	

Nome Classe	RegistrazioneServlet
Descrizione	Questa classe viene utilizzata per la registrazione di un giocatore o di un gestore.
Pre:	**Context** RegistrazioneServlet::doGet(request,response); **Pre:** request.getParameter("username") != NULL && request.getParameter("nome") != NULL && request.getParameter("cognome") != NULL && request.getParameter("e-mail") != NULL && request.getParameter("password") != NULL && request.getParameter("confermaPassword") != NULL && request.getParameter("numeroDiTelefono") != NULL && request.getParameter("dataNascita") != NULL && request.getParameter("nazioneResidenza") != NULL && request.getParameter("provinciaResidenza") != NULL && request.getParameter("capResidenza") != NULL &&

	request.getParameter("cittaResidenza") != NULL && request.getParameter("nomeStruttura") != NULL && request.getParameter("indirizzoStruttura") != NULL && request.getParameter("nazioneStruttura") != NULL && request.getParameter("cittaStruttura") != NULL && request.getParameter("provinciaStruttura") != NULL && request.getParameter("capStruttura") != NULL && request.getParameter("numeroTelefonoStruttura") != NULL;
Post:	**Context** RegistrazioneServlet::doGet(request,response) **Post:** request.getParameter("utente") != NULL;
Invarianti	

Nome Classe	**AreaUtenteServlet**
Descrizione	Questa classe permette accedere all'area utente.
Pre:	**Context** AreaUtenteServlet::doGet(request,response); **Pre:** request.getParameter("gestore") != NULL | | request.getParameter("giocatore") != NULL;
Post:	**Context** AreaUtenteServlet::doGet(request,response) **Post:** request.getParameter("gestore") != NULL | | request.getParameter("giocatore") != NULL;
Invarianti	

Nome Classe	**CreaEventoServlet**
Descrizione	Questa classe permette di creare un evento.
Pre:	**Context** CreaEventoServlet::doGet(request,response); **Pre:** request.getParameter("nome") != NULL && request.getParameter("descrizione") != NULL && request.getParameter("struttura") != NULL && request.getParameter("data") != NULL && request.getParameter("ora") != NULL ;
Post:	**Context** CreaEventoServlet::doGet(request,response) **Post:** request.getParameter("eventi") != NULL ;
Invarianti	

Nome Classe	CronologiaEventiServlet
Descrizione	Questa classe permette di visualizzare la cronologia eventi.
Pre:	**Context** CronologiaEventiServlet::doGet(request,response); **Pre:** request.getParameter("gestore") != NULL
Post:	**Context** CronologiaEventiServlet::doGet(request,response) **Post:** request.getParameter("eventi") != NULL
Invarianti	

Nome Classe	RecensioneServlet
Descrizione	Questa classe permette all'utente registrato di aggiungere una recensione.
Pre:	**Context** RecensioneServlet::doGet(request,response); **Pre:** request.getParameter("username") != NULL && request.getParameter("numeroStelle") != NULL;
Post:	**Context** CreaRecensioneServlet::doGet(request,response) **Post:**
Invarianti	

Nome Classe	EventiRecentiServlet
Descrizione	Questa classe permette all'utente registrato di accedere alla sezione Eventi Recenti.
Pre:	**Context** EventiRecentiServlet::doGet(request,response); **Post:** request.getParameter("giocatore") != NULL ;
Post:	**Context** EventiRecentiServlet::doGet(request,response); **Post:** request.getParameter("eventiRecenti") != NULL ;
Invarianti	

Nome Classe	PartecipaEventiServlet
Descrizione	Questa classe permette all'utente registrato di poter partecipare ad uno o più eventi.
Pre:	**Context** PartecipaEventiServlet::doGet(request,response); **Pre:** request.getParameter("giocatore") != NULL ;
Post:	**Context** PartecipaEventiServlet::doGet(request,response); **Post:** request.getParameter("eventi") != NULL ;
Invarianti	

Nome Classe	RichiesteEventiServlet
Descrizione	Questa classe permette al gestore di gestire le richieste eventi.
Pre:	**Context** RichiesteEventiServlet::doGet(request,response); **Pre:** request.getParameter("gestore") != NULL && request.getParameter("nome") != NULL && request.getParameter("action") != NULL ;
Post:	**Context** RichiesteEventiServlet::doGet(request,response); **Post:**
Invarianti	

Nome Classe	StrutturaServlet
Descrizione	Questa classe permette di visualizzare le strutture da poter selezionare quando si crea un evento.
Pre:	**Context** StrutturaServlet::doGet(request,response); **Pre:**
Post:	**Context** StrutturaServlet::doGet(request,response); **Post:** request.getParameter("strutture") != NULL ;
Invarianti	

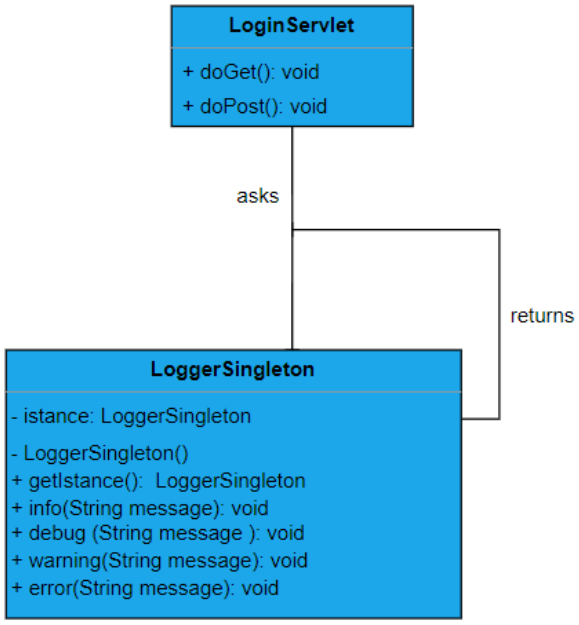
4. Design Pattern e Class Diagram

Di seguito, vengono descritti i design pattern ritenuti idonei all'implementazione del sistema proposto.

4.1 Singleton Design Pattern

Il Singleton è un design pattern di tipo creazionale utilizzato nelle situazioni in cui è necessario istanziare un singolo oggetto della classe di interesse, costituendo un punto di accesso globale per tutte le altre classi del sistema; quindi, possiede uno scope di applicazione e supporta la gestione degli accessi concorrenti dei metodi che offre, attraverso la mutua esclusione. Presenta un unico metodo pubblico getSingleton() utilizzato per creare un'istanza, nel caso non esistesse ancora oppure restituire il riferimento di essa qualora fosse già stata creata. Le variabili di istanza, così come il costruttore, sono private in quanto non è possibile accedervi direttamente.

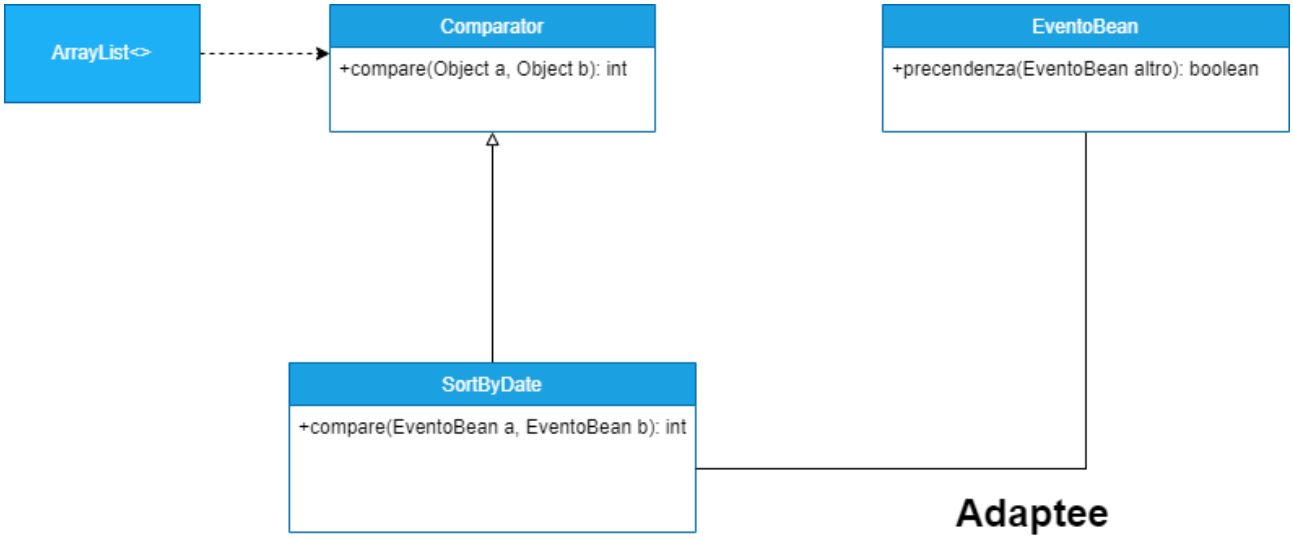
Utilizzando questo design pattern si riesce ad avere un accesso controllato all'unica istanza della classe, si riduce il numero di oggetti condivisi fra le classe e vi è una centralizzazione delle informazioni e dei componenti in un'unica istanza, condivisa però fra più utilizzatori .



Il design pattern verrà utilizzato per gestire l'accesso alla classe LoggerSingleton per appunto effettuare i Log (la classe creerà un file di Log che conterrà tutti i messaggi passati ad esempio in una servler). Qualsiasi classe che intende utilizzare i metodi della classe LoggerSingleto dovrà farlo attraverso l'utilizzo dell'unica istanza della classe presente all'interno del sistema.
Utilizzando Maven, come nel nostro caso, non comparivano in console i Log e quindi questo design pattern, ci è stato molto utile agevolandoci con la complessità del debugging.

4.2 Adapter Design Pattern

L'Adapter è un design pattern che permette di convertire l'interfaccia di una classe in una diversa al fine di permettere a classi diverse con interfacce non compatibili di operare insieme.



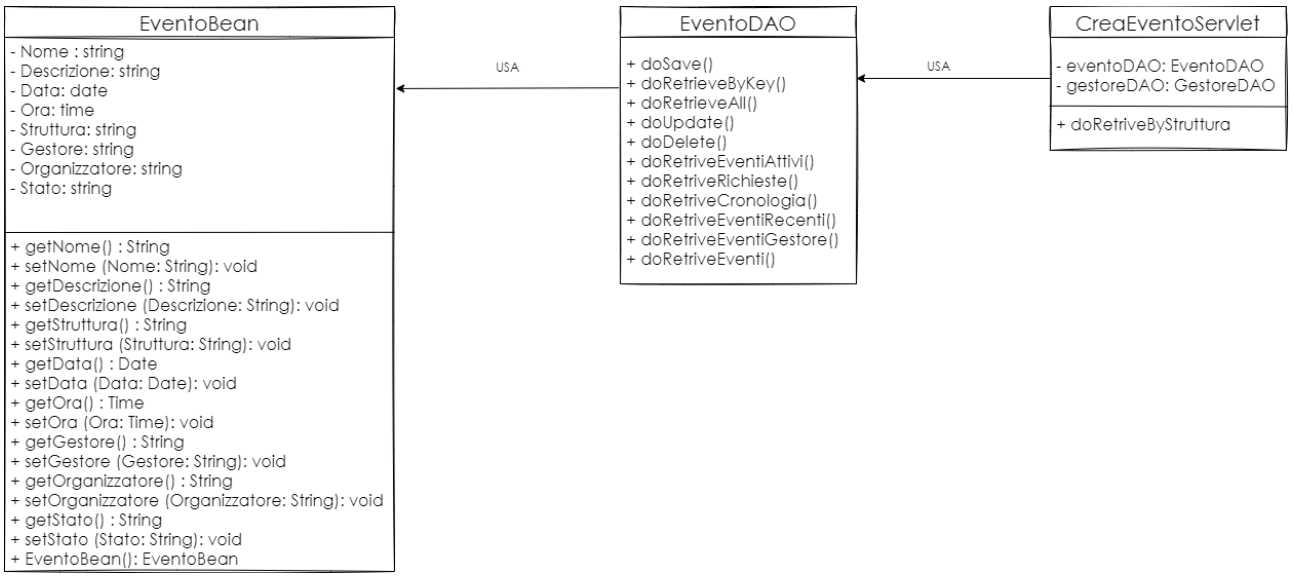
MyBomber farà uso di questo design pattern per ordinare un array di EventoBean in base alla loro data. Verrà definito un nuovo comparatore "SortByDate" che fornisce il metodo compare(), ovvero richiama i metodi di EventoBean.

4.3 DAO

Un DAO (Data Access Object) è un pattern che offre un'interfaccia astratta per alcuni tipi di database. Mappando le chiamate dell'applicazione allo stato persistente, il DAO fornisce alcune operazioni specifiche sui dati senza esporre i dettagli del database. I DAO sono utilizzabili nella maggior parte dei linguaggi e la maggior parte dei software con bisogni di persistenza, principalmente viene associato con applicazioni JavaEE che utilizzano database relazionali.

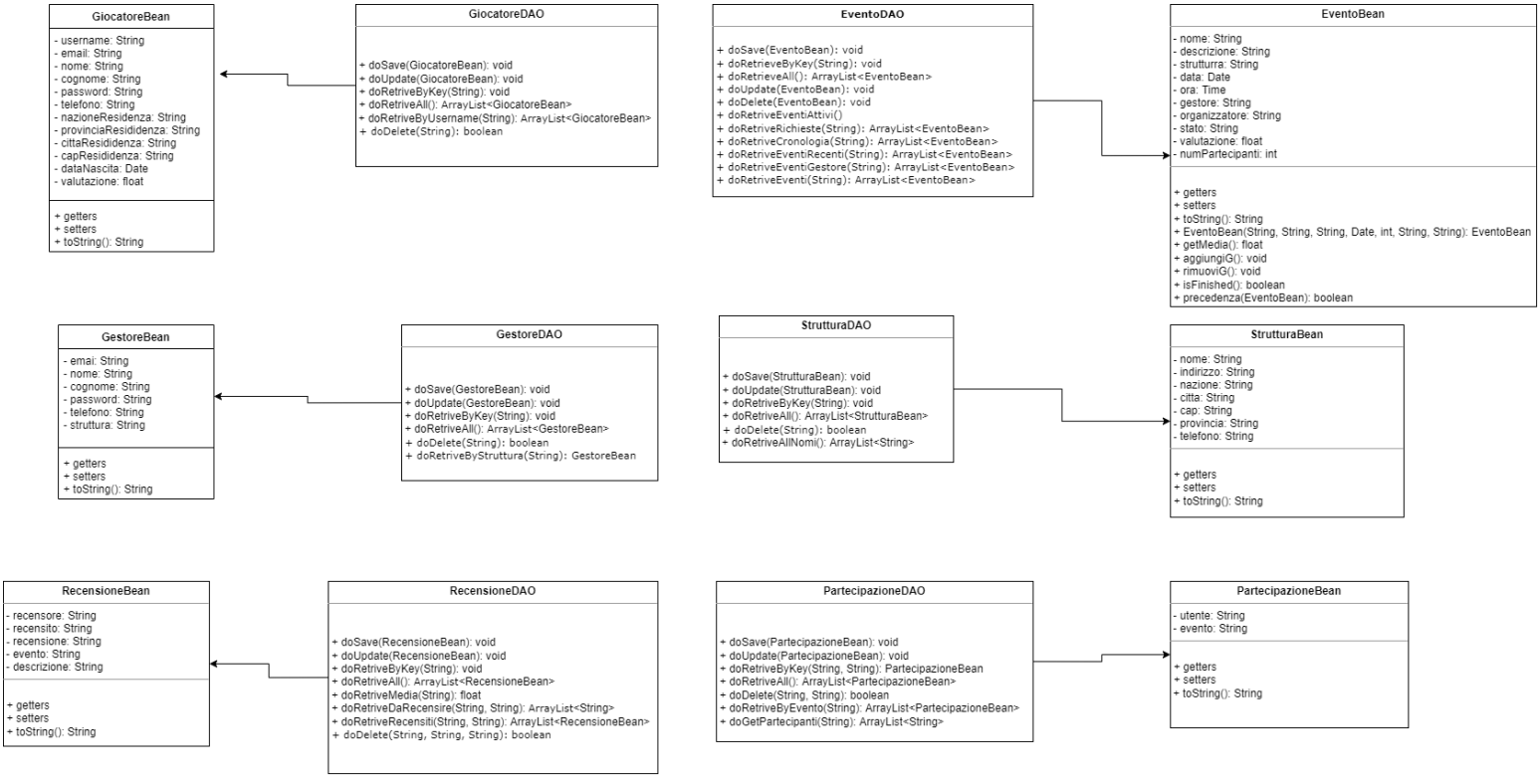
Essendo che MyBomber presenta un database molto vasto, ha bisogno di poter interagire con database in modo rapido e sicuro con numerosi query per quella che è la moltitudine di dati da gestire. Per questo motivo abbiamo usato varie interfacce DAO all'interno del nostro sistema le quali, vengono implementate in maniera del tutto automatica e trasparente per il programmatore.

Di seguito viene riportato un esempio di dove abbiamo utilizzato questo design pattern.



4.4 Class Diagram

4.4.1 Package model



4.4.2 Package control