Object Design Document

Riferimento	
Versione	1.0
Data	20/12/2021
Destinatario	Prof. Gravino
Presentato da	**Team AVA**
Approvato da	

Revision History

DATA	Versione	Cambiamenti	Autori
20/12/2021	0.1	Prima Stesura	[tutti]
21/12/2021	0.2	Aggiunto par 1.4, 1.5	[Alessio, Alessandro]
22/12/2021	0.3	Aggiunte interfacce	[tutti]
3/01/2022	0.4	Aggiunto class diagram finale e glossario	[Alessio, Vincenzo]
13/01/2022	0.5	Corretti design patterns, class diagram, JavaDoc	[tutti]

Sommario

Team members

Nome	Ruolo nel progetto	Email
Vincenzopio Amendola	Team Member	v.amendola15@studenti.unisa.it
Alessio Alfieri	Team Member	a.alfieri33@studenti.unisa.it
Alessandro Rusciano	Team Member	a.rusciano1@studenti.unisa.it

1 Introduzione

Il sistema da realizzare ha come obiettivo la gestione di eventi sportivi (con relative prenotazioni da parte degli utenti) ospitati all'interno delle strutture presenti sul sistema stesso. 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.

1.1 Object Design Goals

Riusabilità: Il sistema deve basarsi sulla riusabilità, attraverso l'utilizzo di ereditarietà e design patterns.
Robustezza: Il sistema deve risultare robusto, reagendo correttamente a situazioni impreviste attraverso il controllo degli errori e la gestione delle eccezioni.
Incapsulamento: Il sistema garantisce la segretezza sui dettagli implementativi delle classi grazie all'utilizzo delle interfacce, rendendo possibile l'utilizzo di funzionalità offerte da diversi componenti o layer sottoforma di blackbox.

1.2 Object Trade-Off

1.2.1 Usabilità vs Funzionalità

Il Sistema dovrà prediligere l'usabilità a discapito delle funzionalità previste nella fase di Analisi in quanto risulta più prioritario fornire un sistema user friendly a discapito di operazioni superficiali.

1.2.2 Costo vs Robustezza

Il Sistema sarà sviluppato in modo robusto a discapito dei costi in quanto essendo di uso sportivo può essere considerato "Mission Critical". Per questo motivo si preferisce sostenere costi maggiori al fine di ottenere un sistema robusto.

1.2.3 Efficienza vs Portabilità

Il Sistema dovrà favorire una maggiore efficienza a discapito della portabilità. Questa scelta nasce dall'esigenza di avere un sistema snello e in grado di eseguire operazioni nel miglior modo e nel minor tempo possibile al fine di suscitare fiducia negli utenti finali del sistema.

1.2.4 Sviluppo Rapido vs Funzionalità

Il Sistema sarà sviluppato con poche funzionalità per favorire uno sviluppo rapido, in quanto sono presenti delle deadline e non è presente abbastanza tempo per implementare anche tutte le funzionalità ritenute meno importanti.

1.2.5 Costo vs Riusabilità

Per il Sistema, essendo realizzato ex novo, non avrebbe senso parlare di riutilizzo di componenti già esistenti. Pertanto non si può ignorare la necessità di sostenere costi maggiori per lo sviluppo.

1.2.6 Tempo di Risposta vs Affidabilità

Il Sistema dovrà garantire una maggiore affidabilità a discapito del tempo di risposta su operazioni critiche in quanto deve essere ridotta al minimo la possibilità di introdurre errori o incongruenze nei dati dovute alla gestione della concorrenza, ecc.

1.3 Components Off The Shelf(COTS)

Il Sistema utilizzerà I seguenti componenti off the shelf:

- ***Bootstrap***, un framework per aiutare lo sviluppo delle interfacce grafiche che utilizza ***HTML, CSS e JS***;

- ***JUnit***, un framework per agevolare lo unit testing per Java;
- ***Apache Tomcat***, un web server con annesso application container per applicazioni Java;

1.4 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. Per la loro costruzione si è fatto riferimento alla convenzione java nota come **Sun Java Coding Conventions** [Sun, 2009].

Link a documentazione ufficiale sulle convenzioni

Di seguito una lista di link alle convenzioni usate per definire le linee guida:

• **Java Sun**: https://checkstyle.sourceforge.io/sun_style.html

• **HTML**: https://www.w3schools.com/html/html5_syntax.asp

1.5 Definizioni, acronimi e abbreviazioni

- **User friendly:** di semplice utilizzo e comprensione per l'utente finale del sistema
- **application container:** è un software che si occupa della gestione completa del ciclo di vita di tutte le classi all'interno e della gestione delle interazioni con l'esterno del sistema;
- **framework**: un insieme di API che svolgono compiti onerosi per lo sviluppatore e che servono per velocizzare e facilitare lo sviluppo del software;
- **style guides**: linee guida per essere conforme ad uno stile che può essere di scrittura, di progettazione, ecc.;
- **UI:** "User Interface", letteralmente interfaccia utente;
- **event-driven:** tipologia di sistema "basato su eventi" dove le operazioni vengono azionate al verificarsi di determinati eventi;
- **HTML:** acronimo di HyperText Markup Language, un linguaggio usato per definire la struttura di una pagina web;
- **CSS:** acronimo di "Cascading Style Sheets", un linguaggio di usato per definire lo stile e la formattazione di una pagina web;
- **JS:** acronimo di JavaScript, un linguaggio di scripting utilizzato nelle pagine web per fornire dinamicità e logica a queste ultime;
- **LowerCamelCase:** tecnica di naming che prevede la scrittura di più parole senza spazi e delimitando l'inizio di una nuova parola con una maiuscola. La prima lettera della prima parola è in minuscolo;
- **UpperCamelCase:** tecnica di naming uguale alla precedente con l'unica differenza riguardo la prima lettera della prima parola che in questo caso è maiuscola;

1.6 Riferimenti

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

- ***Statement of Work;***
- ***Requirements Analysis Document;***
- ***System Design Document;***
- ***Object Design Document;***
- ***Test Plan;***

2. Packages

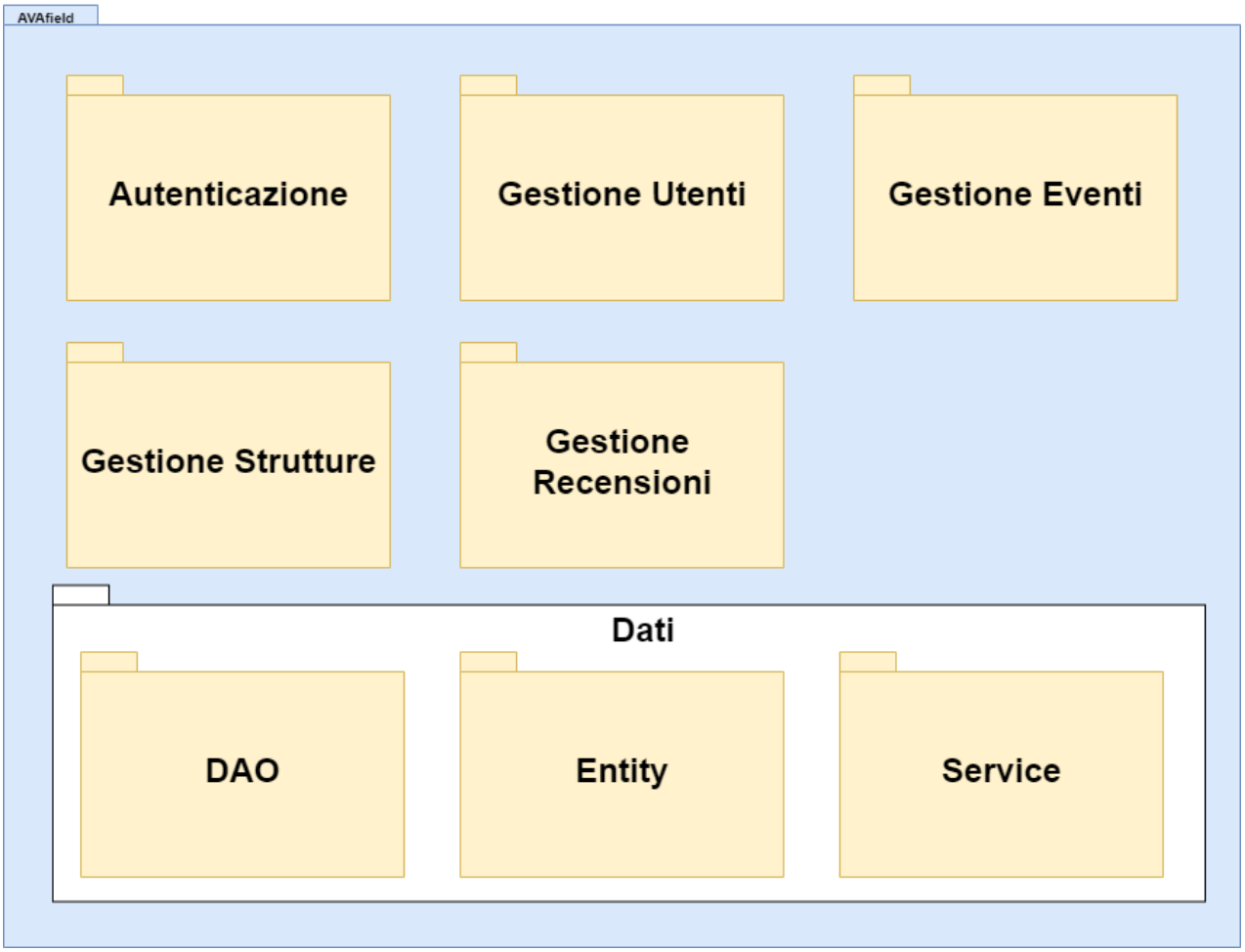
In questa sezione viene mostrata la suddivisione del sistema in package, in base a quanto definito nel documento di System Design. Tale suddivisione è motivata dalle scelte architetturali prese e ricalca la struttura di directory standard definita da Maven.

- ***.idea***
- ***.pom.xml***, contiene tutti i file di configurazione per Maven
- ***src***, contiene tutti i file sorgente
    - ***main***
        - ***java***, contiene le classi Java relative alle componenti Application e Dati
        - ***webapp***, contiene i file relativi alle componenti Interface
            - ***css***, contiene i fogli di stile CSS
            - ***js*** contiene i fogli javascript
            - ***WEB-INF***, contiene i file .jsp
    - ***test***, contiene tutto il necessario per il testing
        - ***java***, contiene le classi Java per l'implementazione del testing
- ***target***, contiene tutti i file prodotti dal sistema di build di Maven

Nella presente sezione si mostra la struttura del package principale di AVAfield. La struttura generale è stata ottenuta a partire da due scelte:

1. Creare un package separato per ogni sottosistema, contenente la classe controller del sottosistema, ed eventuali classi di utilità usate unicamente da esso.
2. Creare un package separato per le classi dei Dati, contenente le classi entity e i DAO per l'accesso al DB. Tale scelta è stata presa vista la presenza del database di AVAfield che prevede relazioni tra le entità. Si è quindi preferito tenere tutto in un package separato e collegato a tutti gli altri package dei sottosistemi.

3. Class Interfaces



Javadoc di AVAfield

All'interno della cartella della documentazione del progetto è possibile consultare da web browser il file *JavaDoc/index.html,* contenente la documentazione del codice. Lo stesso file è reperibile alla repository GitHub navigando */doc/deliverables/JavaDoc/index.html*

Package Gestione Utente

Nome Classe	***UtenteService***
Descrizione	Questa classe consente di gestire le operazioni relative all'Utente
Metodi	+ registrazione(Utente u) + login (String Email, String Pass): Utente + modificaDati (Utente utente) + cancellazioneAccount(Utente utente)

	+ visualizzaUtenti(): List<Utenti>
Invariante di Classe	

Nome Metodo	**+ Registrazione (Utente utente)**
Descrizione	Questo metodo consente di registrare un nuovo Utente
Pre-Condizione	**context**: RegistrazioneService:: registrazione(Utente utente) **pre:** visualizzaUtenti. exists(utente)==false
Post-Condizione	**context**: RegistrazioneService:: registrazione(Utente utente) **post**: visualizzaUtenti. exists(utente)==true

Nome Metodo	**+ login (String Email, String Pass): Utente**
Descrizione	Questo metodo consente di loggare un Utente
Pre-Condizione	**/**
Post-Condizione	**context**: AutenticazioneService::login(email, password) **post**: isUtente(loggedUser) || isAdmin (loggedUser) ==true

Nome Metodo	**+ modificaDati (Utente utente): Utente**
Descrizione	Questo metodo consente di aggiornare l'account di un Utente
Pre-Condizione	**context**: AutenticazioneService::modificaDati (utente) **pre**: visualizzaUtenti.exists(utente)==true
Post-Condizione	**/**

Nome Metodo	**+ cancellazioneAccount (Utente utente): Utente**
Descrizione	Questo metodo consente di cancellare l'account di un Utente
Pre-Condizione	**context**: AutenticazioneService:: cancellazioneAccount (Utente utente) **pre**: visualizzaUtenti.exists(utente)==true
Post-Condizione	**context**: AutenticazioneService ::cancellazioneAccount (Utente utente) **post**: visualizzaUtenti. exists(utente)==false and visualizzaUtenti.size == @pre.visualizzaUtenti.size-1

Package Gestione Eventi

Nome Classe	***EventiService***
Descrizione	Questa classe permette di gestire le operazioni relative agli Eventi
Metodi	+ creaEvento(Evento evento): Evento + eliminaEvento(Evento evento): Evento + modificaEvento(Evento evento) : Evento + partecipaEvento(Evento evento,Utente utente) + visualizzaEventi(): List<Eventi> + findAllUtenti(Evento evento): List<Utente>
Invariante di Classe	

Nome Metodo	**+ creaEvento(Evento evento): Evento**
Descrizione	Questo metodo permette di creare un Evento.
Pre-Condizione	**context**: EventiService::creaEvento(evento) **pre**: visualizzaEventi.exists(evento)==false
Post-Condizione	**context**: EventiService::creaEvento(evento) **post**: visualizzaEventi.exists(evento) == true

	and visualizzaEventi.size == @pre.visualizzaEventi.size+1

Nome Metodo	**+ eliminaEvento(Evento evento): Evento**
Descrizione	Questo metodo permette di eliminare un evento
Pre-Condizione	**context**: EventoService:: eliminaEvento(evento) **pre**: visualizzaEventi.exists(evento)==true
Post-Condizione	**context**: EventoService:: eliminaEvento(evento) **post**: visualizzaEventi.exists(evento)==false **and** visualizzaEventi.size == @pre.visualizzaEventi.size-1

Nome Metodo	**+ modificaEvento(Evento evento): Evento**
Descrizione	Questo metodo permette di modificare un evento
Pre-Condizione	**context**: EventoService:: eliminaEvento(evento) **pre**: visualizzaEventi.exists(evento)==true
Post-Condizione	**/**

Nome Metodo	**+ partecipaEvento(Evento evento,Utente utente)**
Descrizione	Questo metodo permette di far partecipare un Utente ad un Evento.
Pre-Condizione	**context**: EventoService:: partecipaEvento(evento, utente) **pre**: not findAllUtenti(evento).contains(utente) **and** visualizzaEventi.exists(evento)==true
Post-Condizione	**context**: EventoService:: partecipaEvento(evento, utente) **post**: findAllUtenti(evento).contains(utente) **and** findAllUtenti(evento).size == @pre.findAllUtenti(evento).size+1

Nome Metodo	**+ visualizzaEventi(): List<Evento>**

Descrizione	Questo metodo permette di visualizzare tutte gli Eventi
Pre-Condizione	**/**
Post-Condizione	**/**

Nome Metodo	**+ findAllUtenti(Evento evento): List<Utente>**
Descrizione	Questo metodo permette di visualizzare tutti gli Utenti di un Evento
Pre-Condizione	**context**: EventoService:: findAllUtenti(Evento evento): List<Utente> **pre**: visualizzaEventi.exists(evento)==true
Post-Condizione	**/**

Package Gestione Strutture

Nome Classe	***StrutturaService***
Descrizione	Questa classe permette di gestire le operazioni relative alle Strutture
Metodi	+ inserisciStruttura(Struttura struttura) + eliminaStruttura(Struttura struttura) + modificaStruttura(Struttura struttura) + visualizzaStrutture(): List<Strutture>
Invariante di Classe	

Nome Metodo	**+ inserisciStruttura(Struttura struttura): Struttura**
Descrizione	Questo metodo permette di inserire una nuova struttura

Pre-Condizione	**context**: StrutturaService::inserisciStruttura (struttura) **pre**: visualizzaStrutture.exists(struttura)==false
Post-Condizione	**context**: StrutturaService:: inserisciStruttura (struttura) **post**: visualizzaStrutture.exists(struttura) == true **and** visualizzaStrutture.size == @pre.visualizzaStrutture.size+1

Nome Metodo	**+ eliminaStruttura(Struttura struttura): Struttura**
Descrizione	Questo metodo permette di eliminare una struttura
Pre-Condizione	**context**: StrutturaService::eliminaStruttura (struttura) **pre**: visualizzaStrutture.exist(struttura) == true
Post-Condizione	**context**: StrutturaService::eliminaStruttura (struttura) **post**: visualizzaStrutture.exist(struttura) == false **and** visualizzaStrutture.size == @pre.visualizzaStrutture.size-1

Nome Metodo	**+ modificaStruttura(Struttura struttura): Struttura**
Descrizione	Questo metodo permette di modifica una struttura
Pre-Condizione	**context**: StrutturaService:: modificaStruttura (struttura) **pre**: visualizzaStrutture.exist(struttura)==true
Post-Condizione	**/**

Nome Metodo	**+ visualizzaStrutture(): List<Strutture>**
Descrizione	Questo metodo permette di visualizzare tutte le Strutture
Pre-Condizione	**/**
Post-Condizione	**/**

Package Gestione Recensioni

Nome Classe	***RecensioniService***
Descrizione	Questa classe permette di gestire le operazioni relative alle Recensioni
Metodi	+ creaRecensione(Recensione recensione) + eliminaRecensione(Recensione recensione) + modificaRecensione(Recensione recensione) + visualizzaRecensioni(): List<Recensioni> + visualizzaRecensioneByIdStruttura(): List<Recensioni>
Invariante di Classe	

Nome Metodo	**+ creaRecensione(Recensione recensione):Recensione**
Descrizione	Questo metodo permette di creare una nuova Recensione
Pre-Condizione	**context**: RecensioneService::creaRecensione (recensione) **pre**: visualizzaRecensioni.exists(recensione)==false
Post-Condizione	**context**: RecensioneService::creaRecensione (recensione) **post**: visualizzaRecensioni.exists(recensione) ==true **and** visualizzaRecensioni.size == @pre.visualizzaRecensioni.size+1

Nome Metodo	**+ eliminaRecensione(Recensione recensione): Recensione**
Descrizione	Questo metodo permette di eliminare una recensione
Pre-Condizione	**context**: RecensioneService::eliminaRecensione (recensione) **pre**: visualizzaRecensioni.exists(recensione)==true
Post-Condizione	**context**: RecensioniService::eliminaRecensione (recensione)

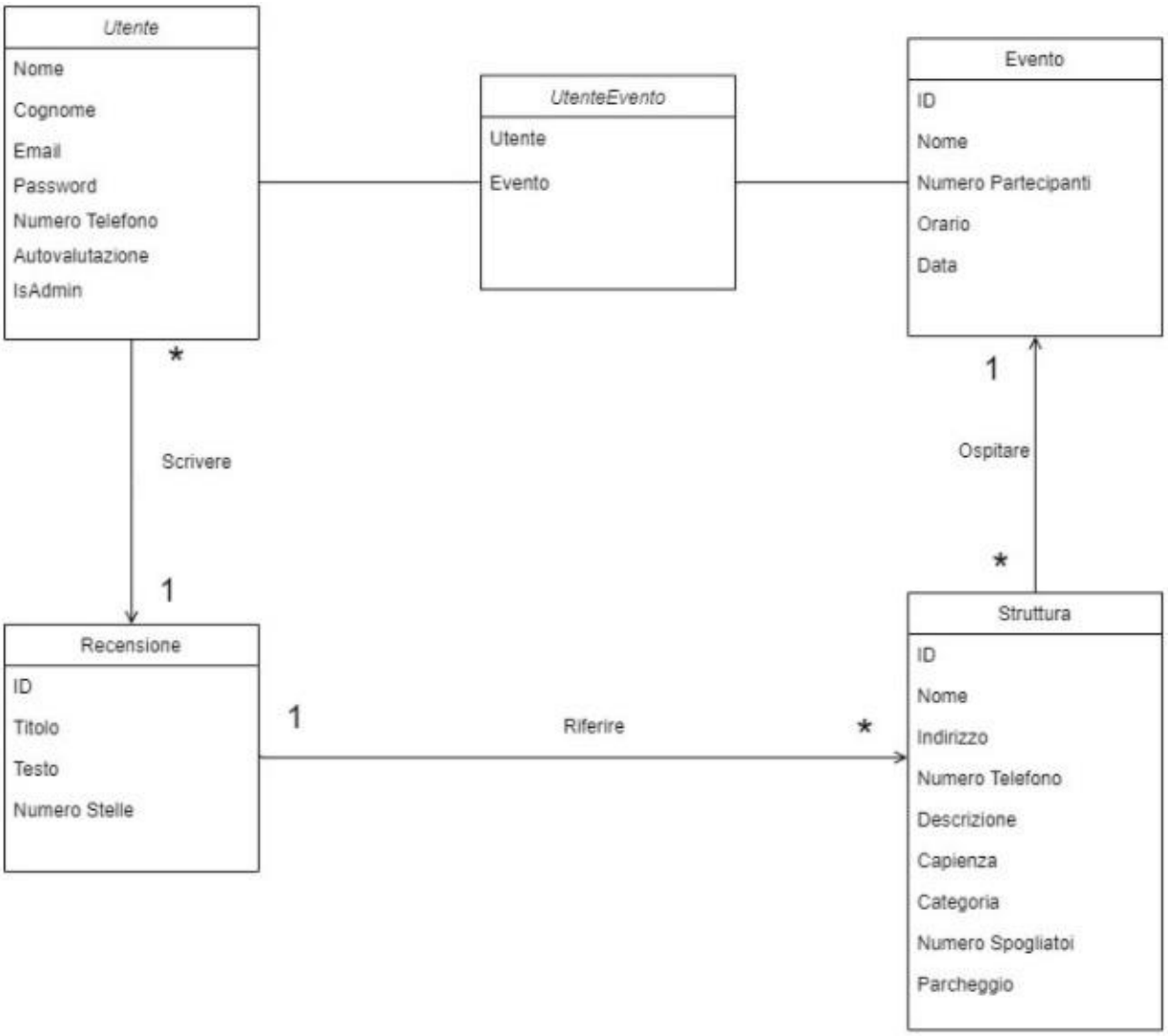
	post: visualizzaRecensioni.exists(recensione) ==false **and** visualizzaRecensioni.size == @pre.visualizzaRecensioni.size-1

Nome Metodo	**+ modificaRecensione(Recensione recensione) : Recensione**
Descrizione	Questo metodo permette di modificare Recensione
Pre-Condizione	**context**: RecensioniService::modificaRecensione (recensione) **pre**: visualizzaStrutture.exist(recensione)==true
Post-Condizione	**/**

Nome Metodo	**+ visualizzaRecensioni(): List<Recensioni>**
Descrizione	Questo metodo permette di visualizzare tutte le Recensioni
Pre-Condizione	**/**
Post-Condizione	**/**

Nome Metodo	**+ visualizzaRecensioneByIdStruttura(): List<Recensioni>**
Descrizione	Questo metodo permette di visualizzare tutte le Recensioni di una Struttura
Pre-Condizione	**/**
Post-Condizione	**/**

4. Class Diagram



5. Design Patterns

In questa sezione si andranno a descrivere i design pattern utilizzati nello sviluppo dell'applicativo di AVAfield. Per ogni pattern si darà:

• Una brevissima introduzione teorica.

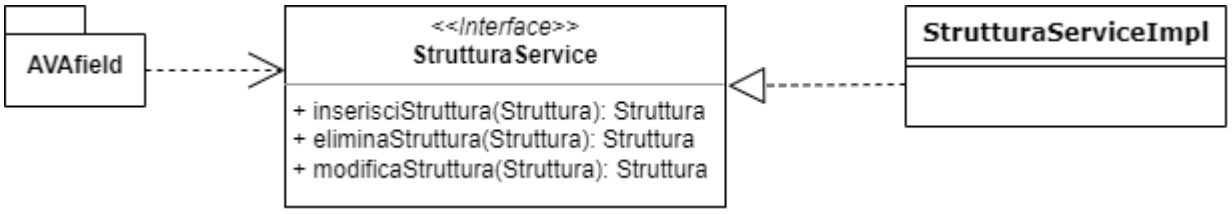
• Il problema che doveva risolvere all'interno di AVAfield.
• Una brevissima spiegazione di come si è risolto il problema in AVAfield.
• Un grafico della struttura delle classi che implementano il pattern.

Façade

Il Facade è un design pattern che permette di accedere a sottosistemi attraverso delle interfacce. In questo modo si può nascondere al sistema la complessità delle librerie, dei framework o dei set di classi che si stanno usando. Si garantisce così un alto disaccoppiamento e si rende la piattaforma più manutenibile e più aggiornabile, poiché basterà cambiare l'implementazione dei metodi dell'interfaccia per implementare le modifiche.

AVAfield sfrutta il design pattern Facade per implementare tutta la sua logica di business e rendere più facile interfacciarsi con essa. Nello specifico utilizza il Facade per ogni suo sottosistema, implementando attraverso delle interfacce che sono usate per accedere ai metodi interni.

Esempio:

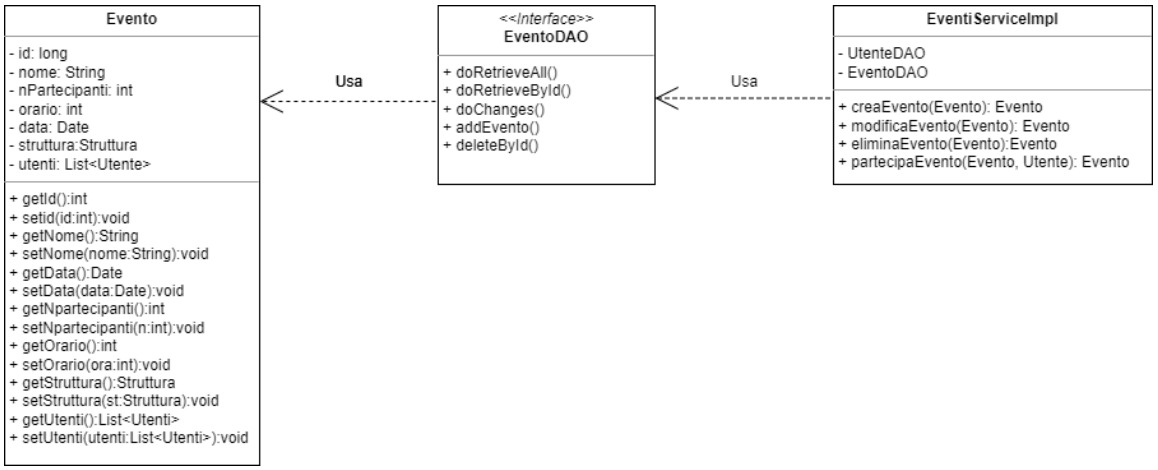


DAO

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.

AVAfield è una web application che punta a gestire le prenotazioni di Eventi sportivi presso strutture e presenta un database, quindi ha bisogno di poter interagire con esso in modo rapido e sicuro con query. Per questo motivo abbiamo usato varie interfacce DAO all'interno del nostro sistema.

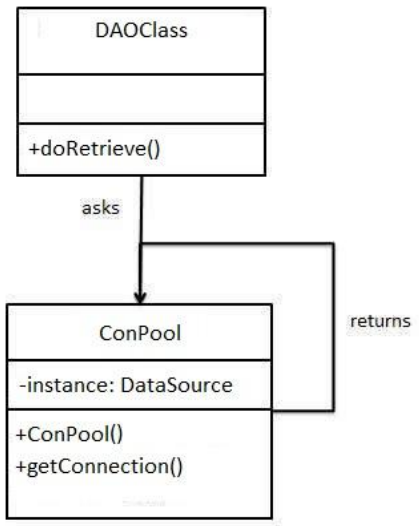
Esempio:



Singleton Pattern

Il singleton è un design pattern creazionale 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. L'implementazione più semplice di questo pattern prevede che la classe fornisca un metodo "getter" statico che restituisce l'istanza della classe (sempre la stessa), creandola preventivamente alla prima chiamata del metodo, e memorizzando il riferimento in un attributo privato anch'esso statico.

AVAfield richiede continui accessi a un database, quindi ha bisogno di poter interagire con la stessa istanza del database stesso.



6. Glossario

- ***Deadline***, ovvero una scadenza;
- ***Off The Shelf***, solitamente preceduto da “Component”, è un qualcosa già pronto all’uso;
- ***DB***, acronimo di DataBase ovvero “Base di Dati” che è un archivio persistente;
- ***SQL***, acronimo di “Structured Query Language”, che è un linguaggio dichiarativo utilizzato per interagire con DataBase di tipo Relazionale;
- ***UI***, acronimo di “User Interface”, ovvero un’interfaccia grafica con la quale un utente può interagire;