Easy Fut5al

Object Design Document

Sommario

1. Introduzione

1.1 Object design trade-offs

Comprensibilità vs Costi

Si preferisce aggiungere costi per la documentazione al fine di rendere il codice comprensibile anche alle persone non coinvolte nel progetto o le persone coinvolte che non hanno lavorato a quella parte in particolare. Commenti diffusi nel codice facilitano la comprensione, di conseguenza migliorare la comprensibilità agevola il mantenimento e anche il processo di modifica.

Prestazioni vs Costi

Si è preferito aggiungere costi a livello di tempo di scrittura del codice per raggiungere una maggiore efficienza nelle prestazioni.

Interfaccia vs Easy-use

La natura del software pensata è caratterizzata da paradigmi per dispositivi Android che tendono a fornire interfacce chiare e intuitive.

Sicurezza vs Efficienza

Si è preferita la sicurezza anziché l'efficienza perché il prodotto tratta dati sensibili e ci è risultato necessario dare risalto all'affidabilità dei dati presentati ed immessi, per l'accesso al sistema e al database sottostante.

Tempo di risposta vs Spazio di memoria

Si è preferito il tempo di risposta a discapito di una maggiore occupazione dello spazio di memoria del database.

1.2 Linee guida per la documentazione dell'interfaccia

1.2.1 Naming

Utilizzare alcune convenzioni sui nomi rende il codice più leggibile e comprensibile da tutti i membri del team, così da poter intervenire facilmente su di esso.

- ***Classi e interfacce***

I nomi delle classi sono nomi la cui iniziale è in maiuscolo. Se un nome è composto da più parole, esse avranno la lettera iniziale in maiuscolo.
Si evita l'uso di abbreviazioni e acronimi per i nomi delle classi, così da non generare confusione.

- ***Metodi***

I metodi sono di solito composti da verbi con lettera iniziale minuscola.

- ***Costanti***

I nomi di costanti vengono indicati da nomi con tutte le lettere in maiuscolo. Se un nome è formato da più parole, esso viene diviso da un underscore.

1.2.2 Commenti

Sono utilizzati nel codice due tipi di commenti, quelli Javadoc, ovvero le aree di testo comprese tra i simboli /* e */, e quelli stile C, cioè sono righe che iniziano per //. Le funzionalità dei commenti permettono ancora di più una conoscenza mirata e precisa di una porzione di codice che non si conosce.

1.2.3 Altre regole

Vi sono altre regolamentazioni per rendere leggibile un codice:

- Nomi di package, classi e metodi devono essere già di per sé esplicativi così da rendere subito l'idea della loro funzionalità.
- Uso di parti standard dei nomi come ad esempio per le Eccezioni si usa la parola finale "Exception".
- Inizializzare sempre le variabili in un unico blocco, che sia all'inizio di una classe o di un metodo, anche con valori di default.

1.3 Design Patterns

Il sistema fa ampio uso di concetti di riuso, i quali sono stati dettati da opportuni design pattern (template di soluzioni a problemi comuni).

Maggior sforzo si è presentato sulla componente Storage.

· ***Pattern DAO***

Data Access Object (DAO) è un pattern architetturale, implementa un meccanismo di accesso richiesto per lavorare con la sorgente di dati, nel nostro caso il DBMS. La component storage fa uso di un'interfaccia con operazioni di ADD, UPDATE, GETALL, REMOVE, GETBYID costruita ad hoc per ogni tabella del DB. Tali operazioni vengono svolte su Java Bean, permettendo un'astrazione dei dati superiore che supera il legame stretto con la nostra base di dati. I nostri beans costruiti sono: Partita, Atleta, Gestore, Campetto, Gioca.

· ***Pattern Abstract Factory***

Trattasi di un creational pattern, svolge un ruolo centrale per lo sviluppo del nostro prodotto Software. Dopo l'applicazione del Pattern DAO, ci ritroviamo di fronte 5 Bean con le rispettive 5 interfacce DAO (PartitaDAO, AtletaDAO, GestoreDAO, CampettoDAO, GiocaDAO). Si noti come ognuna di esse, offra le stesse funzionalità ma con delle interfacce differenti. Poiché sono presenti più "Factory", è possibile usare una classe astratta per ogni tipo di prodotto e un'implementazione di tale classe per ogni tipo di prodotto concreto. In breve, ogni interfaccia DAO diventa sottoclasse di un'unica classe Fut5alDAO. Quest'ultima lavora direttamente su un solo prodotto astratto, il Bean, astrazione di tutti i Java Bean.

· ***Pattern Façade***

A questo punto abbiamo creato un oggetto StorageFacade che viene piazzato al punto più alto del componente Storage. Esso ha il ruolo di fornire un'unica interfaccia a tutte le funzionalità del sottosistema, perciò il pattern forza un accoppiamento debole fra utilizzatore e sottosistema. Façade prende come parametro una stringa che suggerisce l'interfaccia DAO da usare.

Altri importanti design pattern hanno avuto applicazione sul lato Client Android. Il nostro ambiente di lavoro (Android Studio) favorisce lo sviluppo per mezzo di molteplici ***Adapter Pattern***, che si presentano come soluzione ottimale per molti problemi in cui bisogna settare un oggetto con uno speciale oggetto grafico o widget che sia (ossia quando bisogna inglobare esistenti nel sistema). Tra alcuni pattern di tal genere che abbiamo usato ricordiamo:

SectionsPagerAdapter: *per adattare layout scorrevoli nell'actvity.*

ArrayAdapter: *per inserire valori di un Array dentro un ListView.*

CustomAdapter: *per adattare un template grafico di una partita pubblica con i valori di un certo oggetto Json ricevuto dal server.*

EndGameAdapter: *per adattare un template di una partita da valutare con i valori di un array di Json.*

Infine abbiamo notato come la maggior parte delle componenti client svolgessero operazioni comuni per la connessione al server e per lo scambio di dati in modo asincrono, per ogni classe era necessaria un Asynch Task. Tali metodi sono stati generalizzati in **un'unica classe astratta** (ConnectionTask) e per le poche variazioni dovute alle diverse tipologie di dati da scambiare e ricevere è bastato rendere alcuni metodi astratti e riscrivibili da ogni componente che intendesse usare ConnectionTask.

1.4 Definizioni, acronimi e abbreviazioni

EF5: acronimo che fa riferimento al sistema "Easy Fut5al"

RAD: Requirements Analysis Document.

SDD: System Design Document.

ODD: Object Design Document.

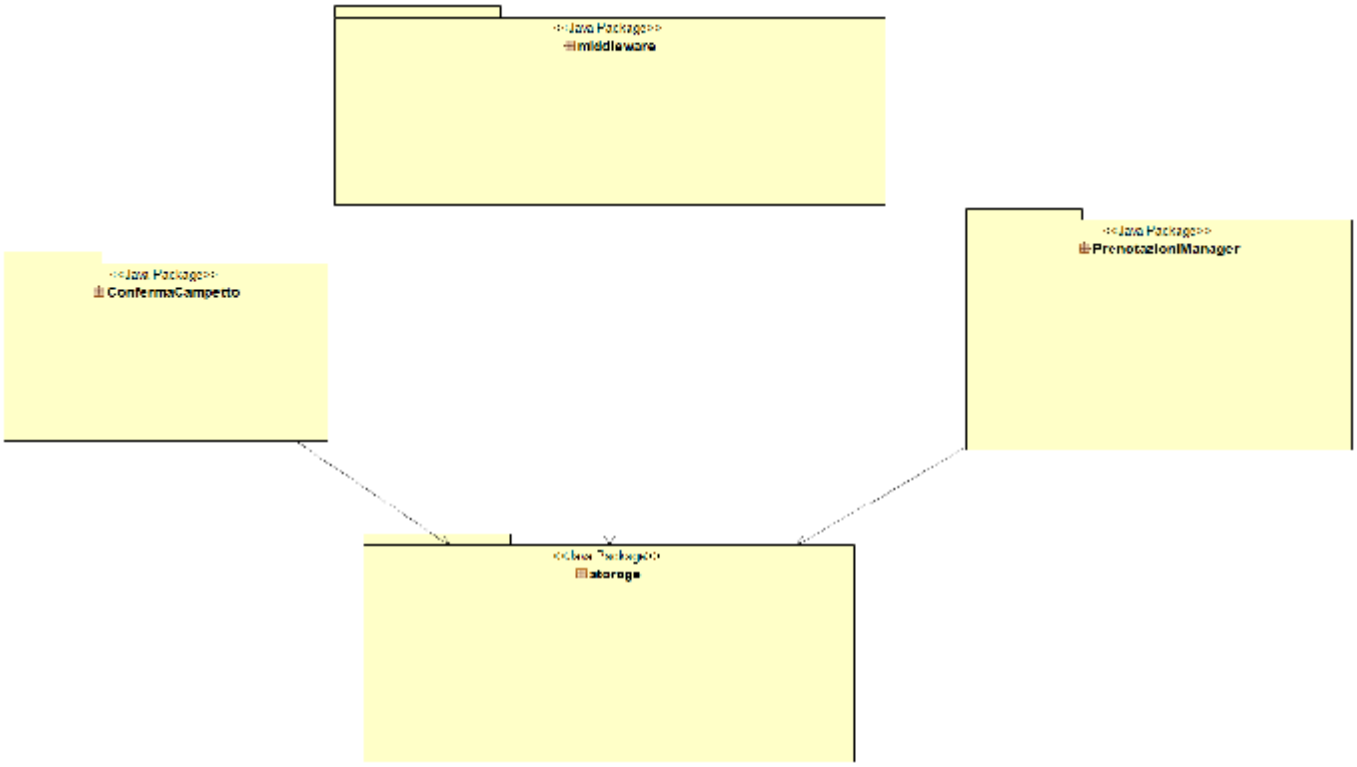
DB: Database.

Design Pattern: Template di soluzioni a problemi comuni.

1.5 Riferimenti

Bernd Bruegge & Allen H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, (3rd edition),Prentice-Hall.
Visual Paradigm for UML
RAD_EasyFut5al

2. Package


middleware
ConfermaCampetto
PrenotazioniManager
storage

3. Interfacce delle classi

3.1 Server-Side package Storage

Atleta: Rappresenta i dati di un atleta con metodi banali quali Constructors, Getter e Setter.

Campetto: Rappresenta i dati di un campetto con metodi banali quali Constructors, Getter e Setter.

Gestore: Rappresenta i dati di un gestore con metodi banali quali Constructors, Getter e Setter.

Partita: Rappresenta i dati di una partita con metodi banali quali Constructors, Getter e Setter.

Gioca: Relaziona una partita ad un atleta rappresentando i dati di un atleta associato ad una partita. Troviamo metodi quali Constructors, Getter e Setter.

StorageFacade: Crea le instanze degli oggetti DAO, quali AtletaDAO, CampettoDAO, GestoreDAO, PartitaDAO e GiocaDAO.
I metodi di questa classe sono:

- public synchronized int Salva(Bean b) : aggiunge un bean dipendente dal tipo scelto (quale Atleta, Gestore, Campetto, Partita e Gioca). Per quanto riguarda Gioca va controllato che il dato da creare non sia ridondante (IsNonRidondante) e che ci siano un minimo di 10 giocatori per una partita (MinDi10)

  Precondizione: (b!=null) && (b.getClass().getName()==(storage.Atleta) || (storage.Campetto) || (storage.Gestore) || (storage.Gioca) || (storage.Partita))

  Postcondizione: (atletaDao).add(b) || (gestoreDao).add(b) || (campettoDao).add(b) ||(giocaDao).add(b) ||(partitaDao).add(b)

- public synchronized boolean IsNonRidondante (Bean b): cerca nel database se ci sia una ridondanza per quanto riguarda il campo Gioca. Se la trova ritorna falso.

  Precondizione: (b!=null)

  Postcondizione: (giocaDao.findRidondance(b)<1) == true

- Public synchronized boolean MinDi10 (Bean b): ha il compito di verificare che in una partita non ci siano più di 10 atleti associati.

  Precondizione: (b!=null)

  Postcondizione: (giocaDao.NumGiocDellaPartita(b)<10) == true.

- public synchronized Collection<Bean> getLista(String type): ritorna una collezione di Bean che desideriamo conoscere (della tipologia Atleta, Gestore, Campetto, Partita e Gioca).

  Precondizione: (type!=””) && ((type.equals(“storage.Atleta”)) || (type.equals(“storage.Campetto”)) || (type.equals(“storage.Gestore”)) || (type.equals(“storage.Gioca”)) || (type.equals(“storage.Partita”)))

  Postcondizione: (Collection<Bean> beans !=null)

- public synchronized Bean getOggetto(String type, int id): ritorna un oggetto Bean che desideriamo conoscere (della tipologia Atleta, Gestore, Campetto, Partita e Gioca).

  Precondizione = (type!=””) && ((type.equals(“storage.Atleta”)) || (type.equals(“storage.Campetto”)) || (type.equals(“storage.Gestore”)) || (type.equals(“storage.Gioca”)) || (type.equals(“storage.Partita”))) && (id>0)

  Postcondizione = Bean b!=null

- public synchronized void aggiorna(Bean b): aggiorna un Bean nel Database.

  Precondizione: (b!=null) && (b.getClass().getName()==(storage.Atleta) || (storage.Campetto) || (storage.Gestore) || (storage.Gioca) || (storage.Partita))

  Postcondizione: b.isUpdate.

- public synchronized void elimina(Bean b): elimina un Bean nel database.

  Precondizione: (b!=null) && (b.getClass().getName()==(storage.Atleta) || (storage.Campetto) || (storage.Gestore) || (storage.Gioca) || (storage.Partita))

  Postcondizione: b.isDelete.

- public synchronized int IDultimaPartita(): ritorna l'id dell'ultima partita caricata.

  Precondizione: //

  Postcondizione: return (partitaDao.lastIdAdd()>0)

3.2 Client-Side

ActivityAtleta: rappresenta la schermata iniziale di un utente loggato correttamente nel sistema come atleta.

- public boolean onCreateOptionsMenu(Menu menu)

  Precondizione: menu!=null

  Postcondizione: menu.isCreated.

- public boolean onOptionsItemSelected(MenuItem item): metodo che gestisce le notifiche arrivate ad un atleta e l'eventuale logout.

  Precondizione: item!=null

  Postcondizione: item.isCreated.

- public void uniscitiPartitaPubblica(View view): questa funziona è associata ad un bottone "unisciti" e permette di unirsi ad una partita se essa non è già al completo.

  Precondizione: (buttonUnisciti.isClicked()) && view!=null

  Postcondizione: //

ActivityAutentificazione: rappresenta la schermata iniziale del sistema che permette ad un utente di loggarsi, oppure di registrarsi o come atleta o come gestore di un campetto.

ActivityGestore: rappresenta la schermata iniziale di un utente loggato correttamente nel sistema come gestore di un campetto. Activity meno complessa rispetto a quella "activityAtleta" in quanto con una sola funzione (quale public boolean onOptionsItemSelected(MenuItem item)), un gestore può vedere il proprio profilo, il proprio campetto e gestisce l'eventuale logout.

ConnectionTask: classe astratta che permette la comunicazione con il lato server.

LoginActivity: activity che viene visualizzata quando un utente dalla schermata principale del sistema, vuole loggarsi. Se il login va a buon fine, reindirizza all'activity relativa allo specifico utente (che sia un atleta oppure un gestore di campetto).

- Private void attemptLogin(): metodo che controlla la bontà dei campi email e password.
- Private boolean isEmailValid(String email): metodo che consente di vedere se la mail inserita è valida.

  Precondizioni: email!=""

  Postcondizioni: email.contains("@")

- Private boolean isPasswordValid(String password) : metodo che gestisce la correttezza della password, nello specifico se è più lunga di 4 caratteri.

  Precondizioni: password!=""

  Postcondizioni: password.length() >4

Profilo: è un frammento che permette di vedere i dati personali (se richiesti) relativi all'utente sia che esso sia un gestore oppure che sia un atleta.

- private void setGestoreProfilo(): se l'utente è loggato come gestore, visualizza il profilo del gestore.

  Precondizione: Iamatleta==false

  Postcondizione: profiloGestore è visualizzato.

- private void setAtletaProfilo(): se l'utente è loggato come atleta visualizza il profilo dell'atleta.

  Precondizione: Iamatleta==true

  Postcondizione: profiloAtleta è visualizzato.

RegistrationActivityAtleta: dall'activity principale, un utente può registrarsi come atleta cliccando il bottone "registrati". Viene reindirizzato a questa activity dove l'utente deve riempire i campi richiesti.

- public boolean controlloCampi(): metodo che controlla i campi dell'activity.

  Precondizione: inputXX!=""

  Postcondizione: (inputXX.getText().toString().length()!=0) && (!(inputPassword.getText().toString().equals (inputConfermaPassword.getText().toString()))

  Nota: inputXX== (inputNome) || (inputCognome) || (inputEmail) || (inputUsername) || (inputPassword) || (inputConfermaPassword) || (inputCitta)

RegistrationActivityGestore: dall'activity principale, un utente può registrarsi come gestore cliccando il bottone "registrati come gestore". Viene reindirizzato a questa activity dove l'utente deve riempire i campi richiesti.

- public boolean controlloCampi(): metodo che controlla i campi dell'activity.

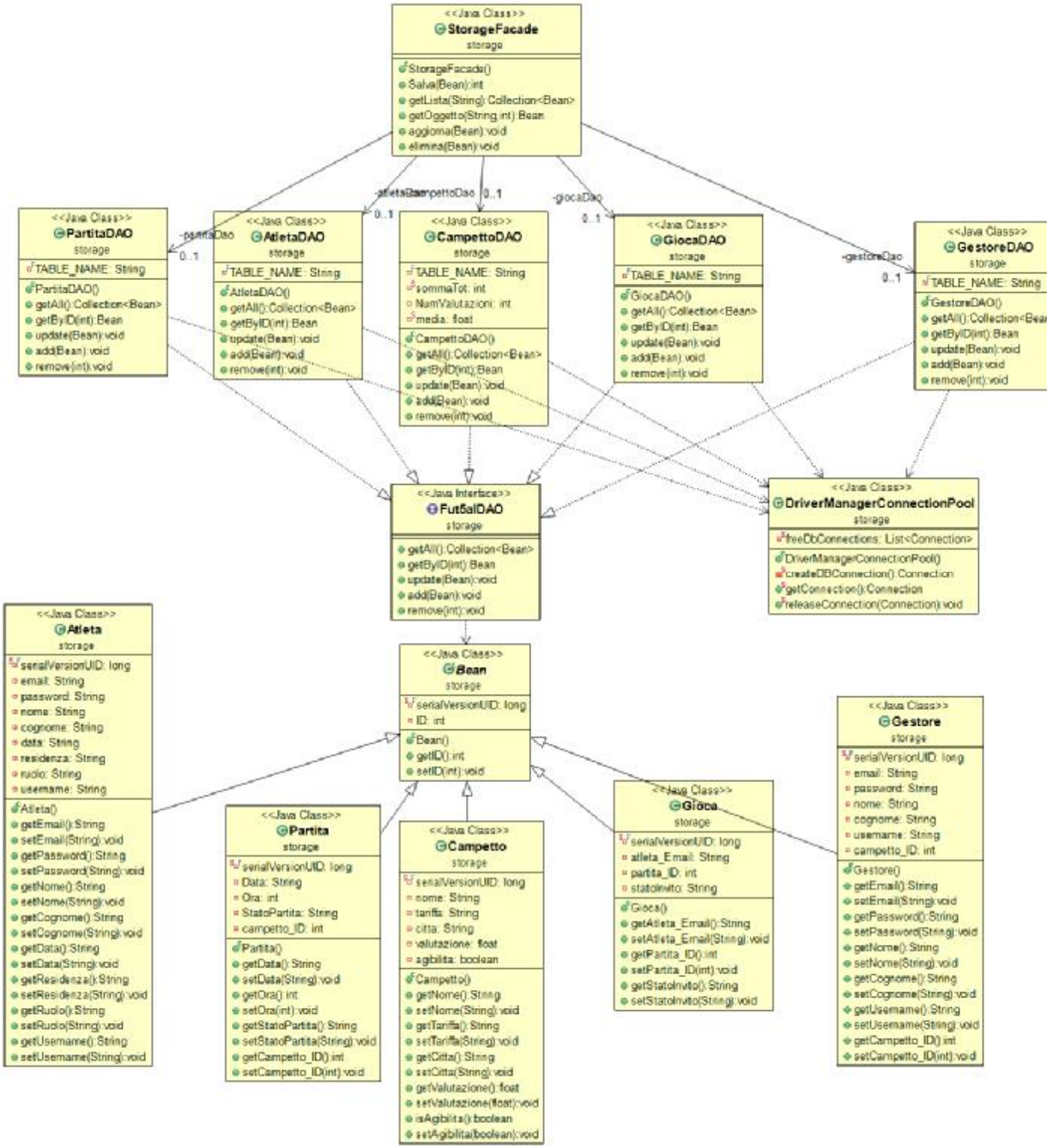
  Precondizione: inputXX!=""

  Postcondizione: (inputXX.getText().toString().length()!=0) && (!(inputPassword.getText().toString().equals (inputConfermaPassword.getText().toString()))

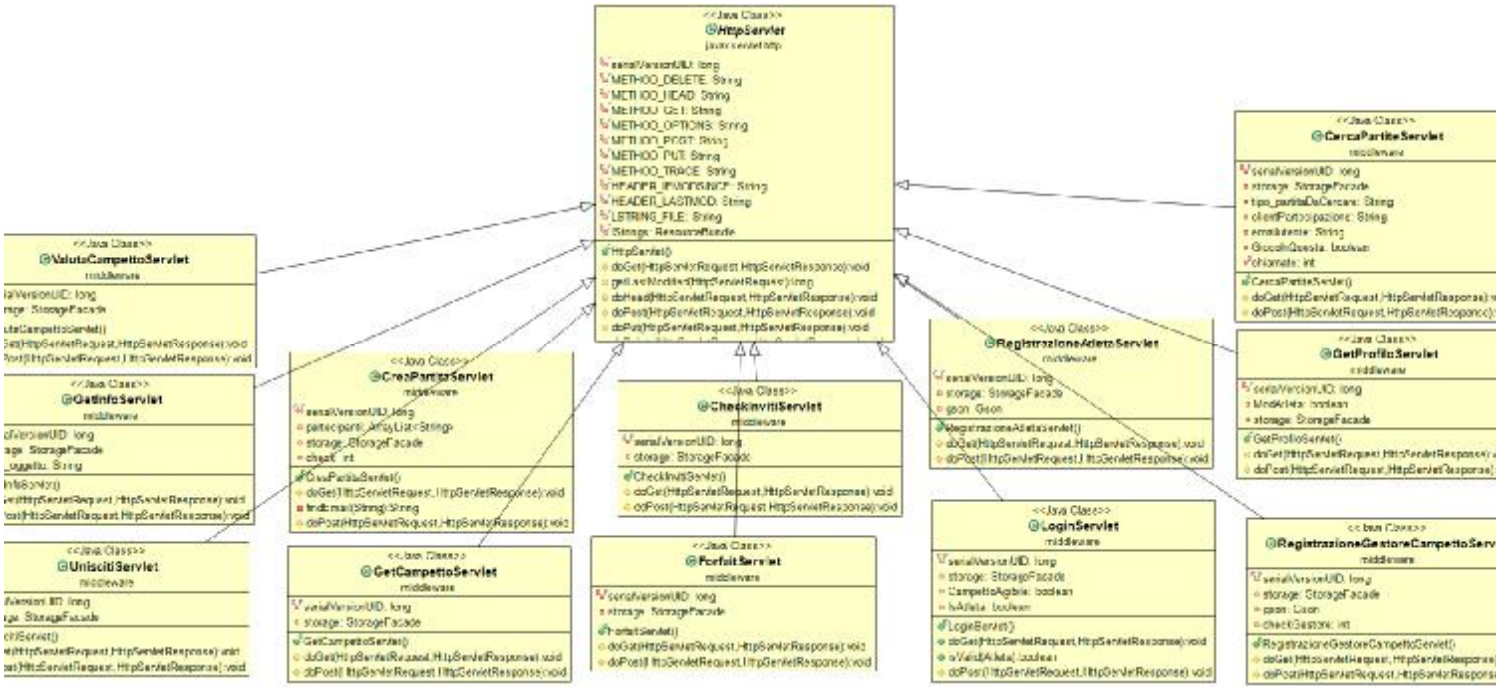
  Nota: inputXX== (inputNome) || (inputCognome) || (inputEmail) || (inputUsername) || (inputPassword) || (inputConfermaPassword) || (inputCittàCampetto) || (inputNomeCampetto) || (inputTariffaCampetto)

4.Class Diagram

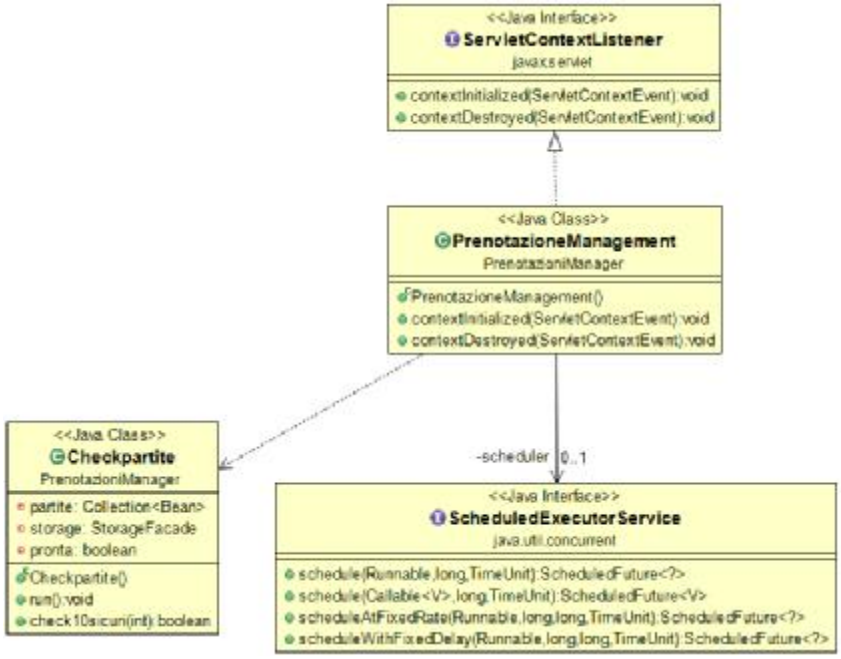
Package Storage



Package Middleware



Package PrenotazioniManager



Client Android Side