Laurea Magistrale in informatica - Università di Salerno
Corso di *Gestione dei Progetti Software* - Prof.ssa F. Ferrucci

Object Design Document

Riferimento	C03_ODD
Versione	1.5
Data	19/01/2021
Destinatario	Prof.ssa Filomena Ferrucci
Presentato da	Giosuè Sulipano
Approvato da	Andrea Massaro

Revision History

Data	Versione	Descrizione	Autori
04/12/2020	0.1	Prima stesura	Tutto il team
15/12/2020	0.2	Stesura Linee Guida	Tutto il team
18/12/2020	1.0	Inserimento dei vari Task e correzioni	Tutto il team
21/12/2020	1.1	Aggiornamento della sezione Package	Tutto il team
11/01/2021	1.2	Aggiornamento tecnologie off-the-shelf	Emma Manzo, Michele Iannucci, Giampiero Ferrara
13/01/2021	1.3	Aggiornamento tecnologie off-the-shelf	Emma Manzo, Michele Iannucci, Giampiero Ferrara
14/01/2021	1.4	Aggiornamento Package e Aggiornamento Class Interfaces	Edoardo Iannaccone, Marco Sica, Umberto Franzese
19/01/2021	1.5	Revisione del documento	Tutto il team

Team

Nominativo	Ruolo
Giosuè Sulipano	PM
Andrea Massaro	Revisore
Edoardo Iannaccone	Sviluppatore
Giampiero Ferrara	Sviluppatore
Emma Manzo	Sviluppatore
Marco Sica	Sviluppatore
Michele Iannucci	Sviluppatore
Umberto Franzese	Sviluppatore

Sommario

1 Introduzione

1.1 Object design trade-offs

Affidabilità vs Tempi di risposta: il sistema garantirà affidabilità sui dati, che saranno sempre consistenti all'interno del nostro database, anche nel caso in cui questa scelta comporti eventuali prolungamenti dei tempi di risposta.

Manutenibilità vs Performance: l'implementazione del sistema favorirà la manutenibilità in modo da permettere allo sviluppatore di apportare modifiche in modo più preciso ed efficace.

Quindi si preferirà una buona strutturazione del codice, puntando ad un basso accoppiamento, anche nel caso in cui questo peggiori le performance.

Leggibilità vs Modificabilità: Il nostro sistema prediligerà leggibilità piuttosto che modificabilità, per cui ogni eventuale modifica implementativa, dovrà sempre attenersi strettamente ai criteri di leggibilità individuati in fase di System Design.

Costi vs Estensibilità: Il rispetto dei costi stabiliti prevarrà sull'estensibilità. Quindi, al fine di rispettare i tempi di rilascio, probabilmente non saranno presenti nella prima release le funzionalità di sistema associate a priorità più basse.

1.2 Componenti off-the-shelf

Per il progetto software che si intende realizzare ci serviremo di diversi componenti *off-the-shelf* ovvero componenti software già sviluppate, ottimizzate, pronte all'uso. Nello specifico, useremo diverse tecnologie, sia il lato back-end e il lato front-end, che ci permetteranno di implementare al meglio la nostra WebApp. Il framework che utilizzeremo per la creazione delle View sarà Bootstrap, una raccolta di strumenti libera, per la creazione di web application. Bootstrap permette di creare modelli di progettazione basati su CSS e HTML, grazie ad essa, sarà possibile gestire il layout della pagina e i suoi diversi componenti, per renderla più interattiva con l'utilizzo di moduli basati su Javascript. In particolare, sarà necessario sfruttare la libreria Chart.js per la visualizzazione delle statistiche raccolte dalla

piattaforma. Verrà utilizzata anche la libreria iText .pdf per generare in automatico il download della scheda di allenamento.
Sarà utilizzato AJAX come strumento di sviluppo software, per gestire i messaggi asincroni all'interno delle pagine. Come web server la scelta, invece, è ricaduta su Tomcat. Il linguaggio di programmazione principalmente utilizzato sarà Java. Tramite JSP (JavaServerPages) e Servlet sarà possibile far comunicare il lato back-end e il lato front-end, permettendo così di avere una pagina dinamica e interattiva. La persistenza dei dati sarà gestita dalla piattaforma cloud Firebase di Google, che ci permetterà di gestire il nostro database runtime.
Tutte le componenti selezionate ed utilizzate sono gratuite ed open source e rispettano i requisiti di costo.

1.3 Linee guida documentazione delle interfacce

Nella fase di implementazione del sistema, gli sviluppatori si dovranno attenere alle linee guida descritte nel seguente paragrafo.

1.3.1 HTML

Il codice HTML sarà presente all'interno delle View del sistema per modellare la struttura dell'interfaccia grafica. La versione di riferimento che verrà utilizzata è la versione 5. Ogni blocco di codice HTML dovrà seguire i seguenti punti:

- Ogni tag di apertura deve essere necessariamente seguito dall'apposito tag di chiusura, eccezione fatta per i tag self-closing (es. <hr>,
, <img>, …).
- Il blocco di codice dev'essere opportunamente indentato.
- L'indentazione del codice deve avvenire tramite tabulazioni (tasto TAB) e non tramite i classici spazi bianchi (tasto BARRA DI SPAZIATURA).
- Il codice dev'essere tutto scritto in lowercase, es. <hr> e non <HR>.
- I tag <script> devono essere posizionati alla fine del file (in genere questi vanno posizionati prima del tag di chiusura </body>).

1.3.2 CSS

Il codice CSS sarà parte delle View per modellare l'aspetto dell'interfaccia grafica del sistema. La versione di riferimento è la versione 3. Ogni blocco di codice CSS dovrà seguire i seguenti punti:

- Il CSS non dev'essere definito all'interno dei tag HTML, bensì dev'essere definito all'interno dei fogli di stile.
- Le regole CSS devono essere scritte su più righe, si dovrà evitare una situazione del genere: .somediv {background: red; padding: 2em; border: 1px solid black;} a favore dello stile seguente.

```
.somediv {
    background: red;
    padding: 2em;
    border: 1px solid black;
}
```

- Non si dovranno riscrivere regole già definite all'interno del framework Bootstrap.
- Utilizzare le shorthand ove possibile.

1.3.3 Java

Il linguaggio Java sarà parte del cuore del sistema, infatti verrà utilizzato per modellarne il comportamento a seguito delle operazioni effettuate dall'utente. La versione di riferimento sarà Java Standard Edition (JavaSE). Il codice Java dovrà seguire i seguenti punti:

- È buona norma utilizzare nomi che siano:
    - Descrittivi
    - Non troppo lunghi
    - Non abbreviati
    - Pronunciabili
- I nomi delle variabili devono essere scritti secondo il Camel Case: devono iniziare con lettera minuscola e le parole seguenti con la lettera maiuscola (per identificare appunto l'inizio di una nuova parola), es. myArray. Le variabili dovranno essere definite all'inizio del blocco di codice.
- Le variabili costanti seguiranno invece la notazione Macro Case: devono utilizzare soltanto lettere maiuscole, separate dal trattino basso: es. ARRAY_SIZE.
- Anche i metodi devono essere scritti secondo il Camel Case. Questi, in genere, sono formati da verbo + nome oggetto: il verbo identifica l'azione da compiere sull'oggetto, es. GetUsername.
- Il codice deve essere provvisto di commenti per facilitarne la lettura e la comprensione. Questi dovranno descrivere la funzionalità oggetto.

- I nomi delle classi e delle pagine devono invece essere scritti secondo il Capital Camel Case: devono iniziare con lettera maiuscola, così come le parole che seguiranno, es. ServletLogin.java.
- I nomi dei package devono essere scritti in Lower Case: devono utilizzare soltanto lettere minuscole: es. account.

1.4 Definizioni, acronimi e abbreviazioni

//

1.5 Riferimenti

Link alle convenzioni utilizzate per la stesura del paragrafo:
https://www.w3schools.com/html/html5_syntax.asp
Link alle convenzioni utilizzate per la stesura del paragrafo:
https://checkstyle.sourceforge.io/sun_style.html
Requirements Analysis Document relativo a questo progetto. Link alla risorsa:

Documento di Statement of Work relativo a questo progetto. Link alla risorsa:

System Design Document relativo a questo progetto. Link alla risorsa:

Link alla libreria utilizzata per la realizzazione dei grafici: https://www.chartjs.org/

Link alla libreria utilizzata per i pdf: https://www.itext.com/en

Link al Javadoc: http://giosuesulipano.it/mypersonaltrainer/

2 Packages

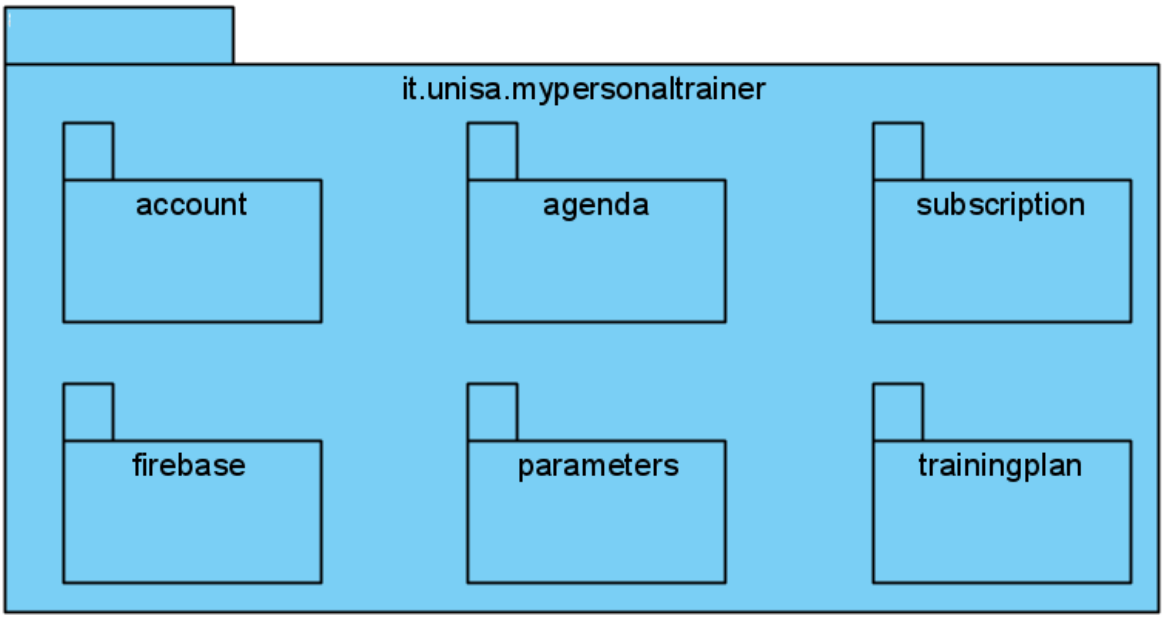
Definiamo come i package vengono suddivisi all'intero del nostro sistema, il tutto condizionato dall'architettura MVC e da quella imposta da Maven.

La nostra directory principale sarà:

- .mvn: contenente i file di configurazione per *Maven*.
- src: contiene i file sorgente
    - main
        - java: classi Java riguardanti le componenti Model e Control
        - webapp: file relativi alle componenti View del sistema
            - css: contiene i file CSS
            - js: contiene i file JavaScript
- pom.xml: file descrittivo, relativo alla gestione delle dipendenze del progetto Maven.

Prestiamo maggiore attenzione alla directory src/main/java ed alla sua struttura in quanto rappresenta il core del progetto.

Mostriamo una visione generale dei package che ne fanno parte:



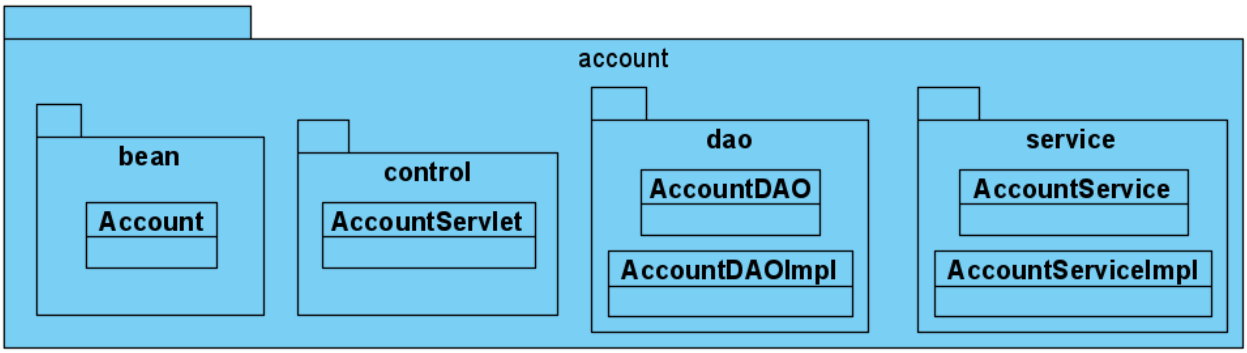
Package:

- account
- agenda
- firebase
- parameters
- subscription
- trainingplan

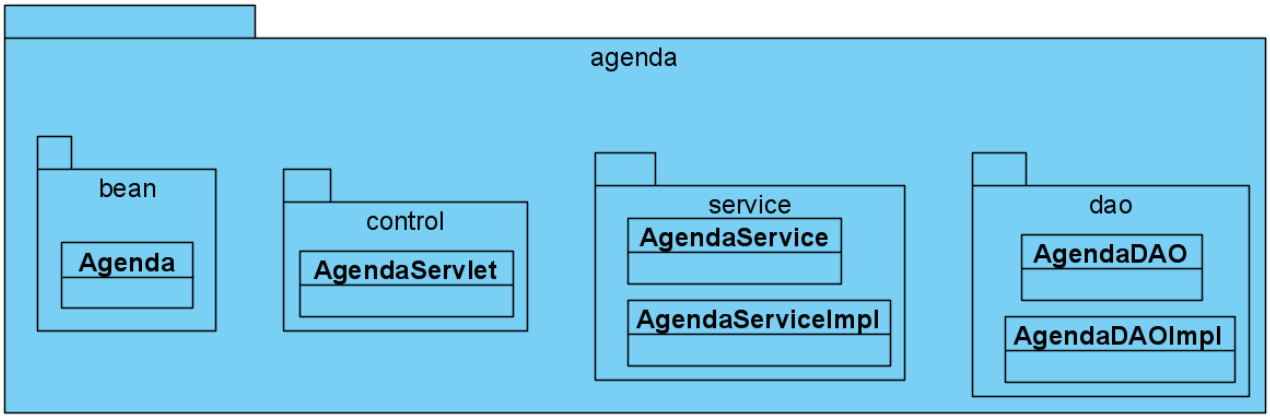
Ogni package contiene la logica di business per la gestione delle componenti e la gestione di esse nel database. Ognuno contiene dei package i quali hanno le seguenti funzionalità:
Package control: contiene le classi che definiscono la sequenza di interazioni con l'utente e la logica di business di una specifica componente. Si precisa che nel package control di ogni sottosistema vi è illustrata solo una classe servlet che rappresenta l'insieme delle servlet effettivamente implementate.
Package dao: interfacce che espongono i metodi CRUD per i dati persistenti e relativa implementazione.
Package bean: contiene le classi che rappresentano gli oggetti del nostro dominio.
Package service: contiene le interfacce che definiscono i metodi di supporto (per servlet e dao) e relativa implementazione.

L'unica eccezione è il package firebase che contiene solamente la classe che permette di connettersi al database.

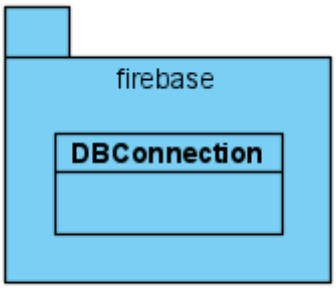
Package account



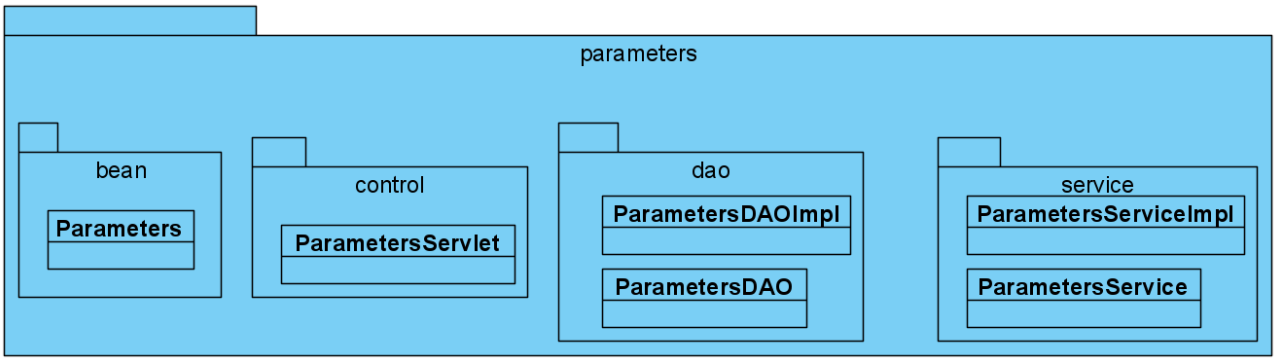
Package agenda



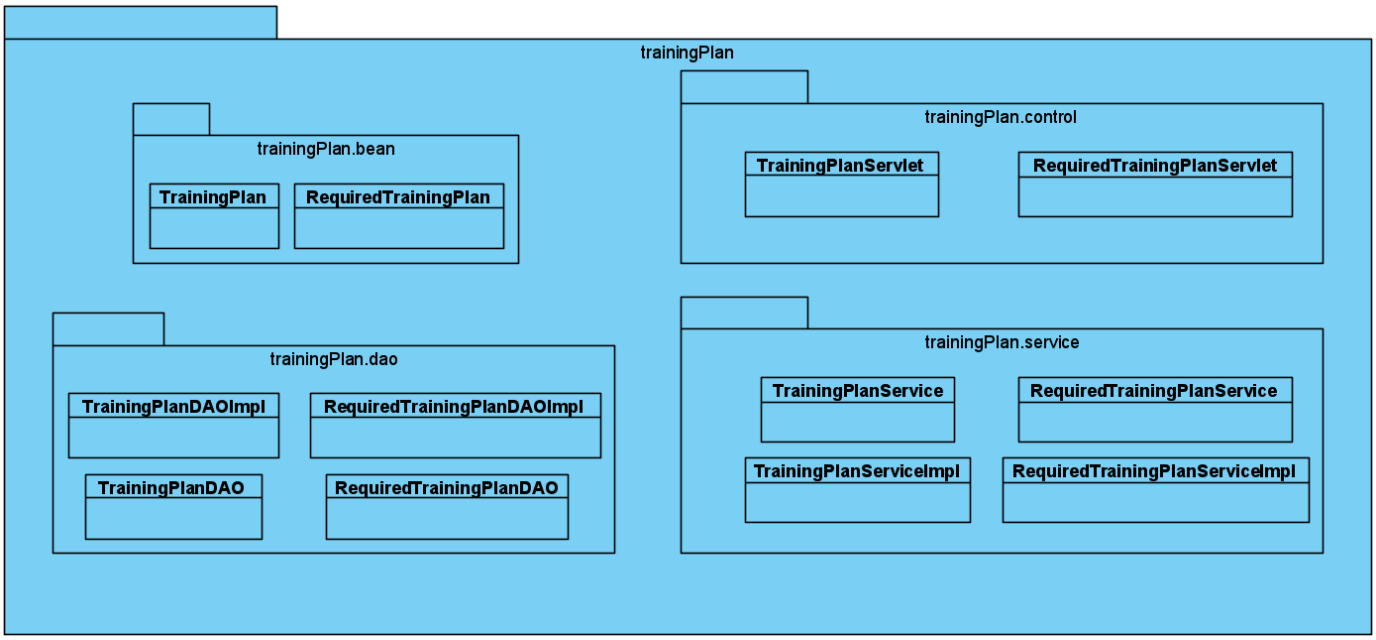
Package firebase



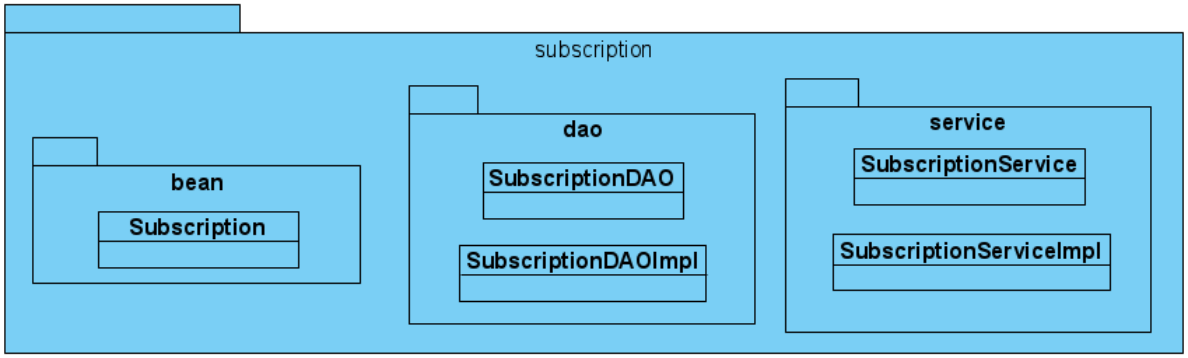
Package parameters



Package TrainingPlan



Package subscription



3 Class interfaces

3.1 Account

Nome	AccountService

Descrizione	Gestisce le funzionalità relative agli account della piattaforma.
Metodi	+loginAccount(String email, String password) : boolean +viewInfoAccount(): ArrayList<Account> +registerAccount(Account utente): boolean +checkCredentials(String clientMail, String newPassword) : boolean +searchAccountByMail(String email) : boolean +getAccountByEmail(String email) : Account +verifyIsAdmin(Account account) : boolean
Invariante	/

+loginAccount(String email, String password) : boolean	
Descrizione	Permette all'utente di effettuare un login nella piattaforma.
Pre - Condizione	**context:** AccountService :: login(e-mail, password) **pre:** (email != null) && (password != null)
Post - Condizione	/

+viewInfoAccount() : ArrayList<Account>	
Descrizione	Ritorna la lista degli account registrati alla piattaforma.
Pre - Condizione	**context:** AccountService :: viewInfoAccount()
Post - Condizione	/

+registerAccount(Account utente) : boolean	
Descrizione	Metodo che permette la registrazione di un nuovo utente alla piattaforma.
Pre - Condizione	**context:** AccountService :: registerAccount(utente) **pre:** (utente != null)
Post - Condizione	/

+checkCredentials(String clientMail, String newPassword) : boolean	
Descrizione	Metodo che permette di controllare la validità dei campi email e password.
Pre - Condizione	**context:** AccountService :: checkCredentials(clientMail, newPassword) **pre:** (clientMail!= null) && (newPassword != null)
Post - Condizione	/

+SearchAccountByEmail(String email) : boolean	
Descrizione	Metodo che permette di vedere se l'account è già presente nel Database.
Pre - Condizione	**context:** AccountService :: searchAccountByEmail(email) **pre:** (email != null)
Post - Condizione	/

+getAccountByEmail(String email) : Account	
Descrizione	Metodo che restituisce l'account registrato alla piattaforma con una specifica email.
Pre - Condizione	**context:** AccountService :: getAccountByEmail(email)

	pre: (email != null)
Post - Condizione	/

+changePassword(String email, String password) : boolean	
Descrizione	Metodo che permette di cambiare la password ad un utente registrato identificato con la propria email.
Pre - Condizione	**context:** AccountService :: changePassword(email, password) **pre:** (email != null) && (password!=null)
Post - Condizione	/

+verifyIsAdmin(Account account) : boolean	
Descrizione	Metodo che permette di controllare se un account è un Admin della piattaforma.
Pre - Condizione	**context:** AccountService :: verifyIsAdmin(account) **pre:** (account !=null)
Post - Condizione	/

Nome	**AccountDAO**
Descrizione	Gestisce le funzionalità CRUD relative agli account della piattaforma.
Metodi	+saveAccount(Account utente) : boolean +findAccountByEmail(email) : Account +updatePassword(String email, String password) : boolean +getAccountDocumentIdByEmail(String email) : String +getAccounts() : ArrayList<Account>
Invariante	/

+saveAccount(Account utente) : boolean	
Descrizione	Salva un nuovo account nel Database.
Pre - Condizione	**context:** AccountDAO :: saveAccount(account) **pre:** (account!=null)
Post - Condizione	/

+findAccountByEmail(String email) : Account	
Descrizione	Restituisce l'account registrato alla piattaforma con una specifica email.
Pre - Condizione	**context:** AccountDAO :: findAccountByEmail(email) **pre:** (email!=null)
Post - Condizione	/

+updatePassword(String email, String password) : boolean

Descrizione	Permette all'utente di cambiare la propria password.
Pre - Condizione	**context:** AccountDAO :: updatePassword(String email, String password) **pre:** (email != null) && (password != null)
Post - Condizione	/

+getAccountDocumentIdByEmail (String emai) : String	
Descrizione	Metodo che restituisce l'id del documento presente nel database associato ad un account con una specifica email.
Pre - Condizione	**context:** AccountDAO :: getAccountDocumentIdByEmail(email) **pre:** (email != null)
Post - Condizione	/

+getAccounts() : ArrayList<Account>	
Descrizione	Ritorna tutti gli account presenti nel database.
Pre - Condizione	**context:** AccountDAO :: getAccounts()
Post - Condizione	/

3.2 Subscription

Nome	**SubscriptionService**
Descrizione	Permette di gestire le varie operazioni eseguibili con un abbonamento.
Metodi	+createSubscription(String customerMail) : void +searchSubscriptionByEmail(String customerMail) : Subscription +checkSubscriptionState(String customerMail) : int +getExpiringSubscription() : ArrayList<Subscription> +getExpiredSubscription() : ArrayList<Subscription> +getActiveSubscription() : ArrayList<Subscriptions> +changeSentNotification(String email) : boolean
Invariante	/

+createSubscription(String customerMail) : void	
Descrizione	Metodo che permette di creare un abbonamento associato ad un cliente della piattaforma.
Pre - Condizione	**context**: SubscriptionService :: createSubscription (customerMail) **pre**:(clientmail != null)
Post - Condizione	/

+searchSubscriptionByEmail(String customerMail) : Subscription	
Descrizione	Restituisce l'abbonamento dell'utente registrato alla piattaforma con una specifica email
Pre - Condizione	**context**: SubscriptionService :: searchSubscriptionByEmail (customerMail) **pre**:(customerMail != null)
Post - Condizione	/

+checkSubscriptionState(String customerMail) : int	
Descrizione	Restituisce lo stato dell'abbonamento relativo all'utente registrato con una specifica email
Pre - Condizione	**context**: SubscriptionService :: checkSubscriptionState(customerMail) **pre**:(customerMail != null)
Post - Condizione	/

+getExpiringSubscription() : ArrayList<Subscription>	
Descrizione	Metodo che restituisce tutti gli abbonamenti che sono vicini alla scadenza
Pre - Condizione	**context**: SubscriptionService :: getExpiringSubscription ()
Post - Condizione	/

+getExpiredSubscription() : ArrayList<Subscription>	
Descrizione	Metodo che restituisce tutti gli abbonamenti scaduti
Pre - Condizione	**context**: SubscriptionService :: getExpiredSubscription ()
Post - Condizione	/

+getActiveSubscription() : ArrayList<Subscription>	
Descrizione	Metodo che restituisce tutti gli Abbonamenti Attivi
Pre - Condizione	**context**: SubscriptionService :: getActiveSubscription ()
Post - Condizione	/

+changeSentNotification(String email) : boolean	
Descrizione	Metodo per cambiare il flag che indica se la notifica è stata inviata o no
Pre - Condizione	**context**: SubscriptionService :: changeSentNotification (email) **pre**:(email != null)
Post - Condizione	/

Nome	**SubscriptionDAO**
Descrizione	Permette di gestire le varie operazioni CRUD di un abbonamento nel database.
Metodi	+ insertSubscription(Subscription sub): Subscription + getSubscriptionbyEmail(String clientmail): Subscription + getAllSubscription(): List<Subscription>

	+getSubscriptionDocumentIdByEmail(String email) : String +updateSentNotification(String email) : boolen
Invariante	/

+insertSubscription(Subscription sub): Subscription	
Descrizione	Metodo che permette di creare ed inserire una Subscription.
Pre - Condizione	**context**: SubscriptionDAO :: insertSubscription (sub) **pre**: (sub != null)
Post - Condizione	/

+getSubscriptionbyEmail(String clientmail): Subscription	
Descrizione	Metodo che permette di cercare una Subscription data la mail del cliente.
Pre - Condizione	**context**: SubscriptionDAO :: getSubscriptionbyEmail (clientmail) **pre**: (clientmail != null)
Post - Condizione	/

+getAllSubscription(): List<Subscription>	
Descrizione	Metodo che permette di cercare tutte le Subscription.
Pre - Condizione	**context**: SubscriptionDAO :: getAllSubscription()
Post - Condizione	/

+getSubscriptionDocumentIdByEmail(String email): String	
Descrizione	Metodo che ritorna l'id del documento relativo all'abbonamneto di un utente con una data email.
Pre - Condizione	**context**: SubscriptionDAO :: getSubscriptionDocumentIdyEmail (email) **pre**: (email != null)
Post - Condizione	/

+updateSentNotification(String email): boolean	
Descrizione	Metodo che permette di cambiare il flag di inviata notifica all'abbonamento associato all'acocunt con mail uguale a quella presa in input.
Pre - Condizione	**context**: SubscriptionDAO :: updateSentNotification (email) **pre**: (email != null)
Post - Condizione	/

3.3 Parameters

Nome	**ParametersService**
Descrizione	Gestisce tutte le operazioni relative ai parametri fisici del cliente
Metodi	+CreateParameters(String weight, String leanMass, String fatMass, String email): Parameters

	+saveParameters(Parameters parameters) +getByMail(String email) : ArrayList<Parameters>
Invariante	/

+createParameters (String weight, String leanMass, String fatMass, String email): Parameters	
Descrizione	Permette di associare dei nuovi parametri ad un cliente, identificato dalla propria mail.
Pre - Condizione	**context:** ParametersService :: createParameters(weight, leanMass, fatMass, email) /
Post	/

+saveParameters (Parameters parameters): boolean	
Descrizione	Rende i parametri persistenti inserendoli nel database.
Pre - Condizione	**context:** ParametersService :: saveParameters(parameters) /
Post	/

+getByMail(String email): ArrayList<Parameters>	
Descrizione	Restituisce tutti i parametri relativi ad un utente, identificato dalla sua email.
Pre - Condizione	**context:** ParametersService :: getByMail(email) /
Post	/

Nome	**ParametersDAO**
Descrizione	Gestisce tutte le operazioni CRUD relative ai parametri fisici del cliente .
Metodi	+insertParameters(Parameters param): boolean +selectByMail(String email): ArrayLisr<Parameters>
Invariante	/

+insertParameters(Parameters param) : boolean	
Descrizione	Questo metodo rende persistenti i parametri nel database .
Pre - Condizione	**Context:** ParametersDAO:: insertParameters(param) **pre: (**param != null)
Post	/

+selectByMail (String email)	
Descrizione	Ritorna tutti i parametri relativi ad uno specifico cliente identificato dalla propria email.
Pre - Condizione	**Context:** ParametersDAO:: selectByMail(email) **pre:** (email !=null)
Post	/

3.4 Agenda

Nome	**AgendaService**
Descrizione	Questa classe contiene le classi di supporto per le operazioni da compiere per il sottosistema Agenda
Metodi	+checkDate(String date): boolean +createAppointment(String date, String time, String mail): boolean +removeAppointment(Appointment appuntamento): boolean +findAppointmentByDate(String date): List<Appointment> +checkAvailability(String data, String time) : boolean +createAvailability(Availability availability): boolean +getAvailabilityByDate(Stirng date): List<Availability> +removeAvailability(Availability availability): boolean +getAvailabilityByDateAndTime(String date, int time): Availability
Invariante	/

+checkDate(String date)	
Descrizione	Permette di controllare la validità della stringa che indica la data
Pre - Condizione	**context:**AgendaService:: checkDate(date) **pre:**(date != null)
Post	/

+createAppointment(String date, String time, String email)	
Descrizione	Controlla la validità dei dati inseriti e rende persistente l'appuntamneto nel database
Pre - Condizione	**context:**AgendaService:: createAppointment(date,time,email)
Post	/

+removeAppointment(Appointment appuntamento)	
Descrizione	Cancella l'appuntamento dal database
Pre - Condizione	**context:**AgendaService:: removeAppointment(appuntamento)
Post	/

+findAppointmentByDate(String date)	
Descrizione	Ritorna tutti gli appuntamenti schedulati in un determinato giorno
Pre - Condizione	**context:**AgendaService:: findAppointmentByDate(date)
Post	/

+checkAvailability(String data, String time)	
Descrizione	Controlla la validità delle stringhe della dell'ora
Pre - Condizione	**context:**AgendaService:: checkAvailability(date,time)
Post	/

+createAvailability(Availability availability)	
Descrizione	Interagisce con il DAO e rende persistente la disponibilità all'interno del database
Pre - Condizione	**context:**AgendaService:: createAvailability(availability)
Post	/

+getAvailabilityByDate(String date)	
Descrizione	Ritorna la lista di disponibilità date per un determinato giorno
Pre - Condizione	**context:**AgendaService:: getAvailabilityByDate(date)
Post	/

+removeAvailability(Availability availability)	
Descrizione	Rimuove la disponibilità dal database
Pre - Condizione	**context:**AgendaService:: removeAvailability(availability)
Post	/

+getAvailabilityByDateAndTime(String date, int time)	
Descrizione	Ritorna la disponibilità indicata per un dato giorno ad una data ora
Pre - Condizione	**context:**AgendaService:: getAvailabilityByDateAndTime(date,time)
Post	/

Nome	**agendaDAO**
Descrizione	Questa classe consente di poter gestire le varie operazioni CRUD relative al sottosistema Agenda
Metodi	+saveAppointment(Appointment appuntamento): boolean +findAppointmentByEmail(String mail): List<Appointment> +findAppointmentByDate(String Date): List<Appointment> +deleteAppointment(Appointment appuntamento): boolean +insertAvailability(Availability availability): boolean +findAvailabilityByDate(String date): List<Availability>
Invariante	/

+saveAppointment(Appointment appuntamento): boolean	
Descrizione	Permette di rendere persistente un appuntamento all'interno del database.
Pre - Condizione	**context**: agendaDAO:: saveAppointment(appuntamento) **pre**: (appuntamento!=null)
Post	/

+findAppointmentByEmail(String mail):List<Appointment>	
Descrizione	Ritorna tutti gli appuntamenti di un dato utente, identificato dalla propria mail.
Pre - Condizione	**context**: agendaDAO:: findAppointmentByEmail(mail) **pre**: (mail!=null)
Post	/

+findAppointmentByDate(String date):List<Appointment>	
Descrizione	Ritorna tutti gli appuntamenti di un dato giorno.
Pre - Condizione	**context**: agendaDAO:: findAppointmentByDate(date) **pre**: (date!=null)
Post	/

+deleteAppointment(Appointment appuntamento): boolean	
Descrizione	Cancella l'appuntamento dal database.
Pre - Condizione	**context**: agendaDAO:: deleteAppointment(appuntamento) **pre**: (appointment!=null)
Post	/

+insertAvailability(Availability availability): boolean	
Descrizione	Rende persistente la disponibilità all'interno del database.
Pre - Condizione	**context**: agendaDAO:: insertAvailability(availability) **pre**: (availability!=null)
Post	/

+findAvailabilityByDate(String date): List<Availability>	
Descrizione	Ritorna tutte le disponibilità date per un determinato giorno.
Pre - Condizione	**context**: agendaDAO:: findAvailabilityByDate(date) **pre**: (date!=null)
Post	/

+deleteAvailability(Availability availability): boolean	
Descrizione	Cancella la disponibilità dal database.
Pre - Condizione	**context**: agendaDAO:: deleteAvailability(availability) **pre**: (appointment!=null)
Post	/

+findAvailabilityByDateAndTime(String date, int time): Availability	
Descrizione	Ritorna l'oggetto availability che rappresenta la disponibilità data per un dato giorno ad una data ora.
Pre - Condizione	**context**: agendaDAO:: findAvailabilityByDateAndTime(date,time) **pre**: (date!=null) && (time!=null)
Post	/

3.5 TrainingPlan

Nome	**TrainingPlanService**
Descrizione	Questo package racchiude tutte le operazioni disponibili per la scheda di allenamento.
Metodi	+getTrainingPlans(String email): List<TrainingPlan> +checkExercise(String exercise, String repetitions, String series, String recoveryTime): boolean +createTrainingPlan(TrainingPlan): boolean
Invariante	/

+getTrainingPlans(String email): List<TrainingPlan>

Descrizione	Metodo che restituisce tutti i traningPlan assegnate ad un utente con email indicata.
Pre-Condizione	**Context:** TrainingPlanService:: getTrainingPlans(email)
Post	/

+checkExercise(String exercise, String repetitions, String series, String recoveryTime): boolean	
Descrizione	Metodo che controlla la validità delle stringhe che componogno il TrainingPlan.
Pre-Condizione	**Context:** TrainingPlanService:: CheckExercise(exercise,repetitions,series,recoveryTime)
Post	/

+createTrainingPlans(TrainingPlan trainingPlan): boolean	
Descrizione	Metodo che richiama il dao creando il TrainingPlan.
Pre-Condizione	**Context:** TrainingPlanService:: createTrainingPlans(trainingPlan)
Post	/

Nome	**TrainingPlanDAO**
Descrizione	Questo classe permette di gestire le query relative all'oggetto TrainingPlan.
Metodi	+getTrainingPlansByEmail(String email): Collection<TrainingPlan> +insertTrainingPlan(TrainingPlan tp): boolean
Invariante	/

+getTrainingPlansByEmail(String email): Collection<TrainingPlan>	
Descrizione	Metodo che ritorna tutte i TrainingPlan associati ad un utente identificato da una data email.
Pre	**context:** TrainingPlanDAO:: getTrainingPlansbyEmail(email) **pre:** (email != null)
Post	/

+insertTrainingPlan(TrainingPlan tp): boolean	
Descrizione	Metodo che rende persistente un TrainingPlan all'interno del database
Pre	**context:** TrainingPlanDAO:: insertTrainingPlan(tp) **pre:** (tp != null)

Post	/

+deleteTrainingPlan(String email): boolean	
Descrizione	Metodo che cancella tutti i TrainingPlan associati ad un utente identificato da una email data
Pre	**context:** TrainingPlanDAO:: deleteTrainingPlan(email) **pre:** (email != null)
Post	/

Per approfondimenti, visionare il Javadoc.

4 Design patterns

In questa sezione vengono illustrati i vari design pattern scelti e nello specifico come agiscono e risolvono la problematica individuata.

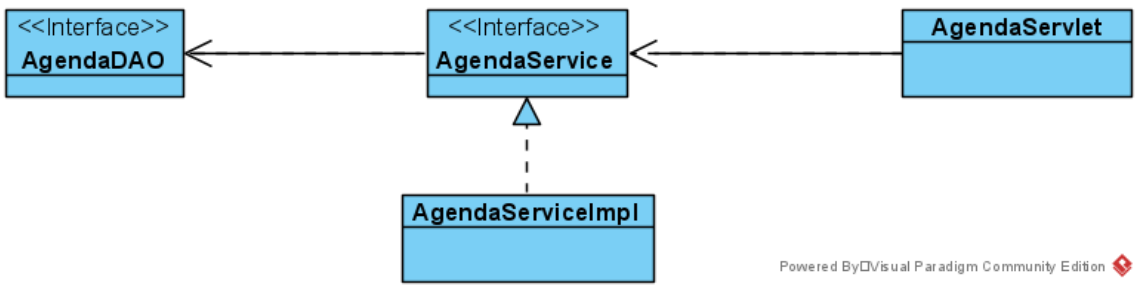
4.1 Façade

Descrizione del problema: Il Façade è un Design Pattern strutturale che fornisce un'interfaccia unificata per accedere ai sottosistemi.

Di solito il client accede direttamente ai servizi di un sottosistema, costringendo la business logic ad essere altamente accoppiata ai dettagli implementativi, complicando la chiarezza del codice ed il suo mantenimento. Infatti, eventuali modifiche alle funzionalità di un sottosistema impatterebbero in modo diretto anche sul client che lo utilizza.

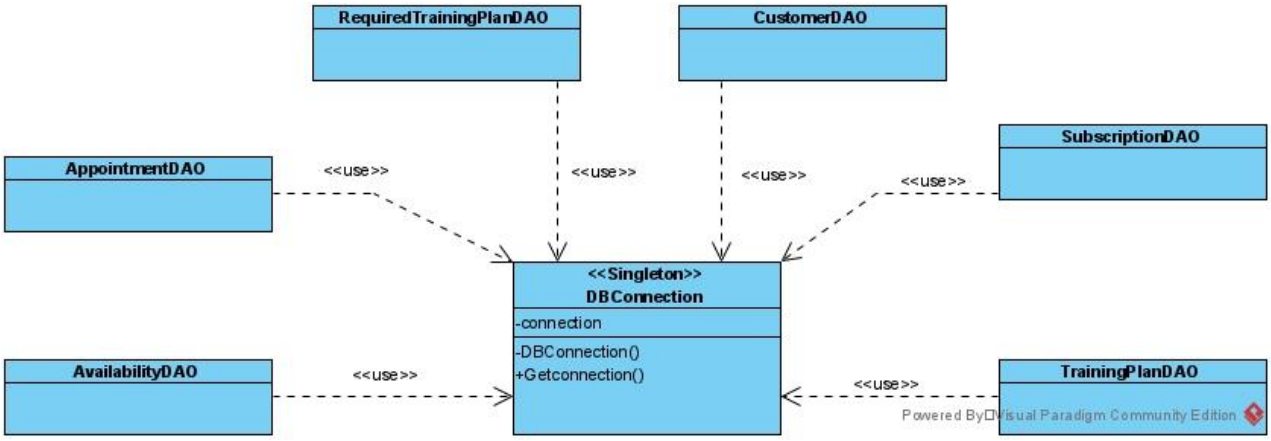
Soluzione: Ogni sottosistema avrà quindi una propria interfaccia pubblica, ad esempio AgendaService, che esporrà tutte le funzionalità fornite e realizzate dal componente Agenda.
Tale interfaccia sarà poi implementata da AgendaServiceImpl.
Questa classe conterrà tutta la logica implementativa dei vari servizi offerti, per cui il chiamante avrà solamente bisogno di istanziarla ed utilizzare il metodo desiderato, senza dover conoscere come è stato implementato effettivamente.

Conseguenze: Le interfacce di alto livello espongono i vari servizi offerti da un sottosistema, rendendolo più facile da utilizzare e nascondendo dettagli implementativi o dipendenze a chi deve utilizzarlo.
Le interfacce si interpongono quindi tra i vari componenti di un sistema, facilitando la comunicazione e diminuendo l'accoppiamento.
L'utilizzo di questo design pattern rafforza l'adozione dei design goals di leggibilità e modificabilità individuati, poiché apporta una buona pulizia e organizzazione del codice, favorendone la manutenzione.



4.2 Singleton

Descrizione del problema: una delle problematiche riscontrate nel nostro progetto, è l'eccessiva istanziazione della classe relativa alla connessione al database.
Soluzione: Il Singleton, design pattern creazionale, ha lo scopo di garantire che venga creata un'unica istanza di una determinata classe. Dunque, l'istanziazione avverrà solamente alla prima chiamata della classe.
Conseguenze: I principali vincoli riguardanti tale scelta riguardano l'impossibilità di sfruttare l'ereditarietà per la classe, quindi, potrebbe esserci difficoltà in alcune scelte di testing.



4.3 Data Access Object

Descrizione del problema: Utilizziamo questo pattern per snellire il codice dei beans e astrarre la logica di business dalla gestione della persistenza dei dati, così da non avere un alto accoppiamento tra essi.
Soluzione: Tramite un'interfaccia DAO si ha l'accesso al Layer di Persistenza, nascondendo all'utente i dettagli di implementazione dei metodi.
Inseriamo inoltre un nuovo layer che si occupa solo della gestione dei dati persistenti, così da avere una gestione centralizzata solo per i dati persistenti.
Conseguenze: Gli oggetti di business non essendo a conoscenza dell'implementazione del DAO garantiscono una possibile migrazione verso nuovo database, comportando modifiche solo al layer di persistenza.
Si risolvono così problemi dovuti a futuri cambiamenti.
Inoltre, migliora la leggibilità e la trasparenza del codice, grazie al basso accoppiamento tra i Layer.

La centralizzazione della gestione dei dati, in un livello sperato, rende più efficiente la manutenzione e la gestione dell'applicazione.