My Workout Buddy

Relazione per “Programmazione ad Oggetti”

Mattia Vandi, Nicola Piscaglia, Lorenzo Pacini

26 febbraio 2015

Indice

Capitolo 1

Analisi

1.1 Requisiti

In Italia sempre più giovani e adulti si dedicano all'attività fisica per mantenersi in forma, mantenere un benessere psicofisico liberando la mente dallo stress accumulato durante la giornata. Il fitness richiede costanza e dedizione alla disciplina per questo va praticata seguendo le giuste indicazioni e tenendo traccia del proprio rendimento e dei progressi raggiunti. L'obiettivo di questa applicazione è di aiutare gli utenti a monitorare i propri miglioramenti, fornendo consigli e programmi di allenamento adeguati alle proprie capacità ed esigenze.

Per perseguire gli obiettivi prefissati abbiamo svolto un'attenta analisi dei requisiti e abbiamo deciso si sviluppare le seguenti funzionalità:

- Creazione di un allenamento personalizzato da parte dell'utente sulla base delle proprie capacità e caratteristiche fisiche
- Calcolo dell'indice di massa corporea (BMI)
- Esposizione di grafici per
    - Tener traccia dei valori del proprio fisico
    - Tener traccia dell'aumento della propria forza
- Aggiornamento costante da parte dell'utente dei propri risultati
- Possibilità di programmare il proprio allenamento settimanale

1.2 Modello del dominio

MyWorkoutBuddy dovrà essere in grado di gestire l'allenamento dell'utente attraverso la creazione di schede di allenamento (routine) personalizzate composte da uno o più allenamenti. Ogni allenamento viene solitamente associato ad un giorno della settimana ed è composto da uno o più esercizi. Ogni esercizio ha un insieme di serie di ripetizioni di numero finito, ogni serie definisce il numero di esecuzioni consecutive dell'esercizio che l'utente dovrà compiere per tale serie. Il numero di ripetizioni può variare tra le serie che sono solitamente intervallate da un tempo di recupero a discrezione dell'utente. Ad ogni esercizio può essere associato un attrezzo da utilizzare.

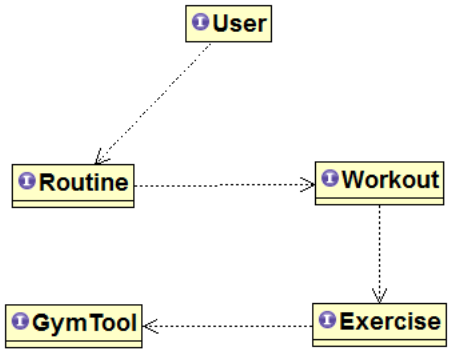


Figura 1: Rappresentazione del dominio del problema

Capitolo 2

Design

2.1 Architettura

Nella progettazione architetturale del sistema ci siamo prefissati di realizzare le varie entità in modo più possibile indipendente per rendere l'applicazione più flessibile, manutenibile e scalabile.
Per conseguire tale obiettivo abbiamo valutato di utilizzare il pattern architetturale Model-View-Controller in quanto permette di separare bene gli aspetti di dominio, controllo e presentazione.
La logica di controllo dell'applicazione è affidata esclusivamente ad una entità che riceve i comandi dell'utente (attraverso la View) e li attua modificando lo stato degli altri due componenti.
La "business logic" del sistema è affidata all'entità di modello che fornisce metodi per permettere la manipolazione da parte dell'entità di controllo.
Il Controller dialoga con la parte di vista reperendo i dati immessi dall'utente e fornisce metodi per la sua manipolazione. La comunicazione tra View e Controller è realizzata attraverso il pattern Observer che permette alla View di modificare il proprio stato su richiesta del controller, in modo indipendente dagli aspetti implementativi delle funzionalità esposte dal Controller. Lo stesso discorso vale per le implementazioni fornite dall'entità di presentazione, le quali sono nascoste all'entità di controllo. In questo modo la presentazione può essere modificata senza nessuna conseguente modifica della parte di controllo.

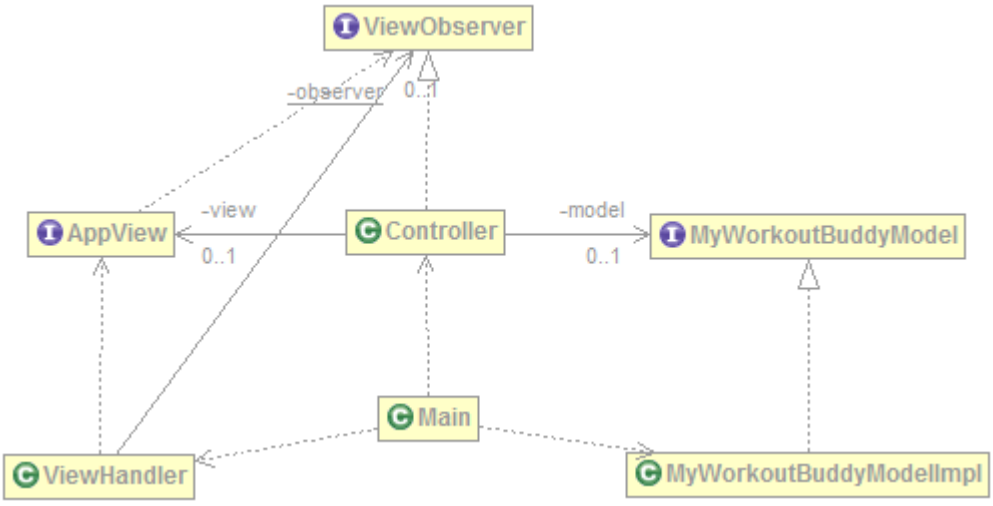


Figura 2: Schema architetturale dell'applicazione. In questo software, un `Controller` *è un osservatore di* `AppViews` *ed è responsabile anche di gestire le interazioni con* `MyWorkoutBuddyModel`*. Si fa presente che* `MyWorkoutBuddyModel` *è l'entry-point del modello e non il modello completo. Lo stesso discorso vale per* `AppViews`*, in quanto non è altro che l'entry-point della logica di presentazione.* `Main` *è il componente responsabile di creare ogni istanza di model, view e controller e di connetterli correttamente tra loro.*

2.2 **Design in dettaglio**

Suddivisione in package

L'applicazione è stata divisa in tre sottoparti logiche, come descritto nell'architettura: Model, Controller e View.
Non avendo alcun dominio, abbiamo deciso di utilizzare il prefisso dell'Università, cui abbiamo aggiunto la sigla del corso: it.unibo.oop. Inoltre, abbiamo aggiunto un suffisso che indentifica questo progetto, per cui tutti i package saranno prefissi da: it.unibo.oop.myworkoutbuddy.
Abbiamo diviso l'applicazione in tre sottoparti logiche in architettura, la divisione in package le rispecchia: model, controller, view.
Il controller è stato suddiviso in due sottopackage `.controller.validation` per gestire la validazione dell'input da parte dell'utente mentre il sottopackage `.controller.db` è stato utilizzato per l'interazione con la base di dati.
Essendo la parte di view composta di molti sorgenti, ho suddiviso alcuni file di .view in sottopackage: .view.factory, .view.handlers. e .view.strategy.
Questa divisione in package della parte di View permette una più facile navigazione tra classi distinte in interfacce delle funzionalità delle View, gestione degli eventi (Handlers), la creazione di oggetti e finestre, strategie di controllo e layout.
Le risorse, invece, sono state posizionate in una folder a parte, divisa anch'essa in package.
Per quanto concerne la View la suddivisione scelta presenta quattro package contenenti rispettivamente le immagini di background, le icone, i file di struttura delle varie View e i fogli di stile.

2.2.1 **Model (Pacini Lorenzo)**

Descrizione MODEL

Il model organizza ed elabora i dati relativi all'attività di uno o più utenti, realizzando le seguenti funzionalità. Inserimento delle informazioni comuni a tutti gli utenti quali: - definizione del corpo umano (Body) in termini di parti (muscoli) e zone di interesse (busto, gambe, ...). - lista e caratteristiche degli attrezzi disponibili nella palestra. Inserimento di informazioni specifiche per ogni utente: - dati dell'utente e relativo account, - schede di allenamento (Workout) dell'utente come lista di esercizi - lista delle attività (Routine) con i relativi valori ottenuti per ogni esercizio Calcolo delle statistiche sulle prestazioni (Score) e sui tempi di allenamento a livello di - singolo allenamento, considerando tutti gli esercizi previsti dalla scheda di allenamento - singolo utente considerando tutte le attività di allenamento svolte. Calcolo degli andamenti (nel periodo di allenamento) dei coefficienti relativi al corpo dell'utente: BMI (Body Mass Index), BMR (Basal Metabolic Rate) e LBM (Lean Body Mass). Calcolo delle statistiche globali sull'utilizzo degli attrezzi da parte di tutti gli utenti che hanno frequentato la palestra (funzionalità disponibile solo con più utenti).

Scelte progettuali (Struttura delle classi)

Il model è organizzato con la seguente struttura di classi: Classi base: Account, Person, Body, BodyData, GymTool, Classi semplici per la gestione degli elementi base Classi composte: User, Exercise, Workout, Routine Classi di composizione per la gestione delle aggregazioni fra i dati User si compone di Accout e Person Exercise si compone di GymTool Workout si

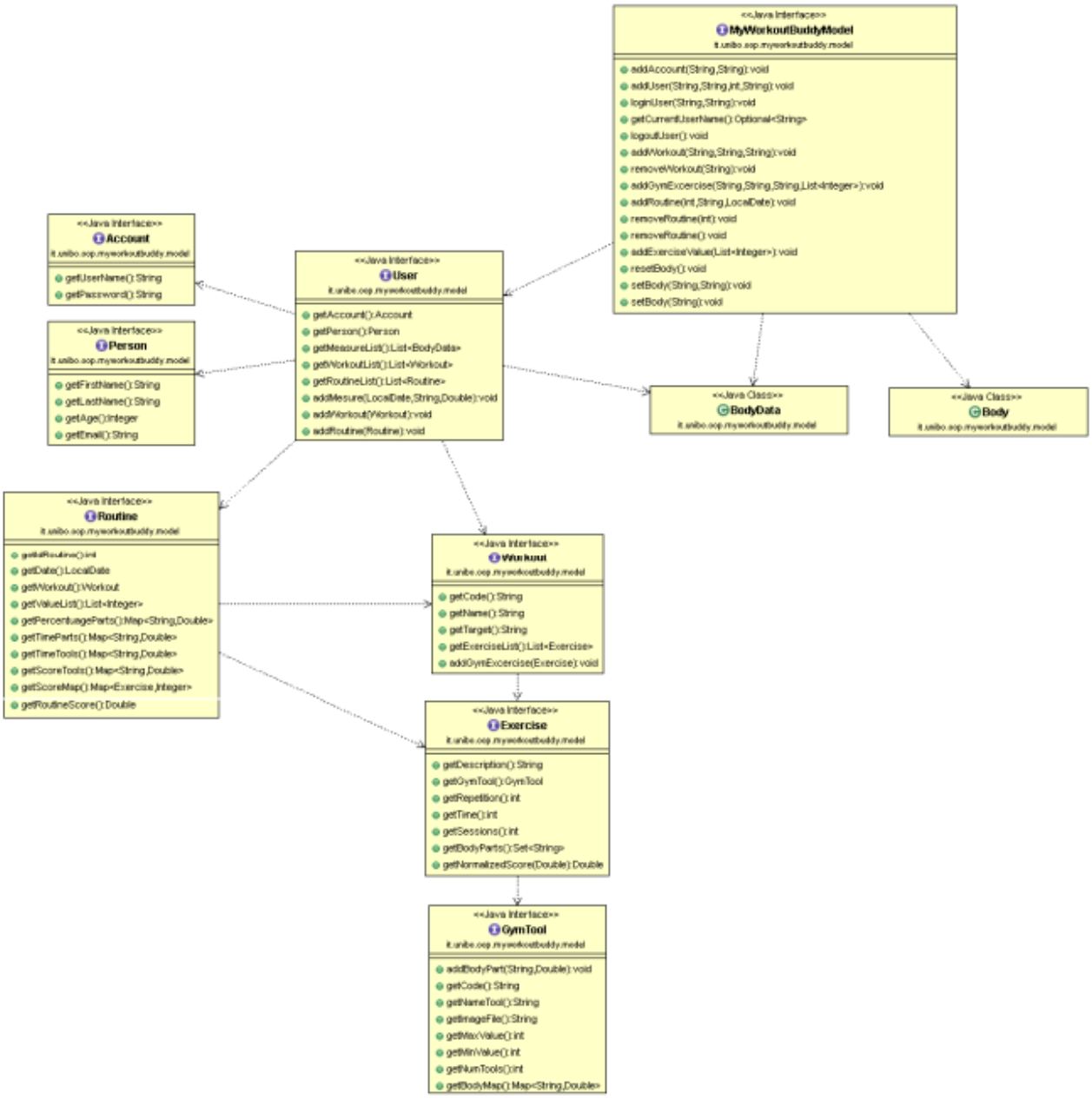
compone di List Routine si compone di un Workout Classi di gestione: ManageUser, ManegeWorkout Classi per la gestione degli elementi correnti Per semplificare il caricamento dei dati e l'interazione, il model utilizza dei riferimenti correnti a cui, una volta impostati, sono riferite le funzioni richieste. Come riferimenti correnti sono gestiti: currentAccount, currentUser, currentWorkout, currentRoutine, currentBodyData. Queste classi hanno anche permesso di rifattorizzare il codice e limitare la dimensione delle altre classi. Inoltre tali classi permettono di realizzare il pattern Adapter verso il controller. Classe Model: MyWorkoutBuddyModelImpl Classe model per l'applicazione definisce l'interfaccia del model verso il controller.

Classi Test:

MainTestModel : Classe di prova del Model, con caricamento di dati e output di tutte le statistiche su console.

MainJUnitTest: Classe per il test automatizzato JUnit.

Model: Architettura generale



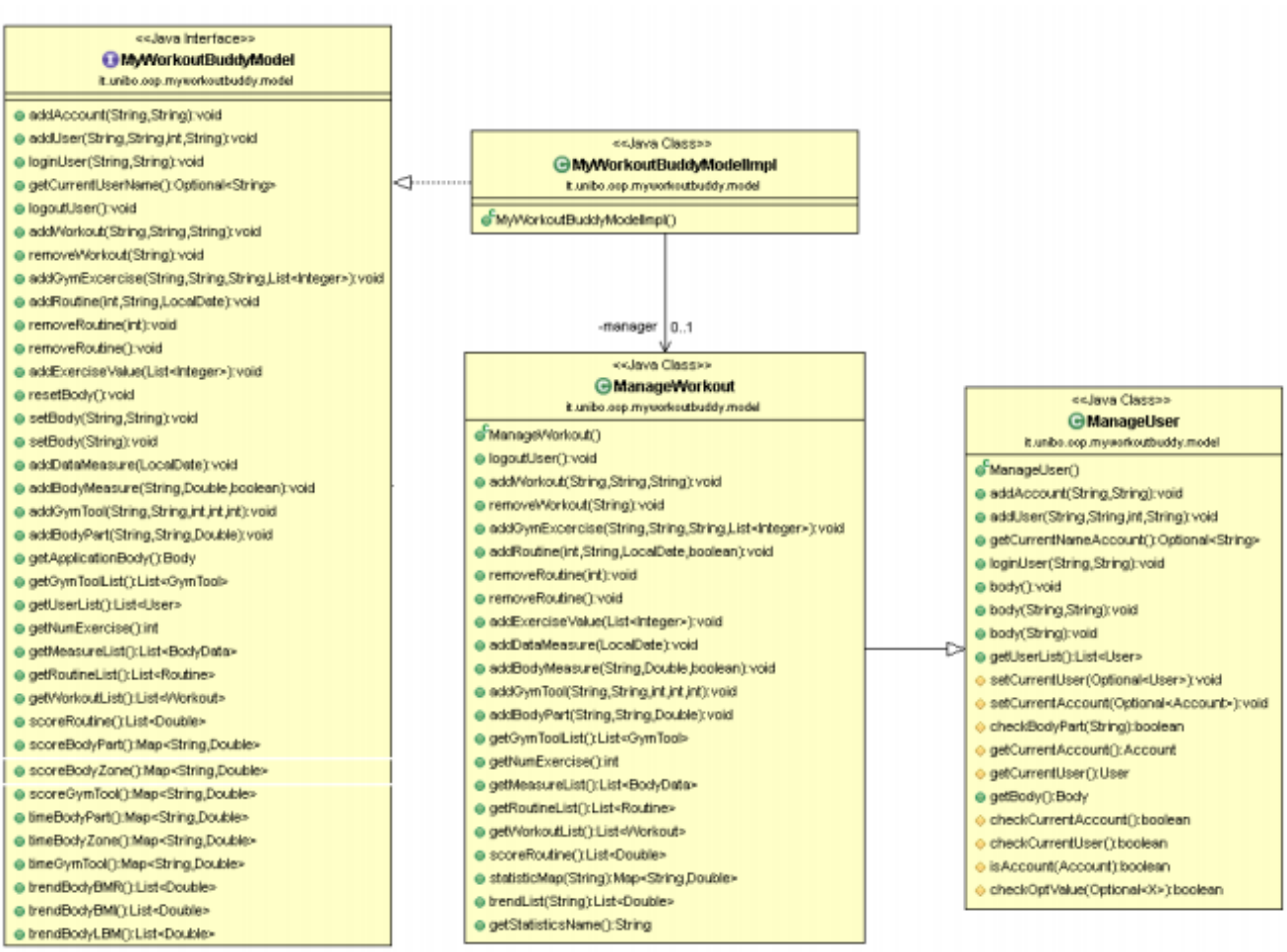
Pattern utilizzati

- Strategy, metodi di map, filter e merge usati principalmente in classe User per il calcolo delle statistiche - Builder: costruzione classe GymToolImpl ed ExerciseImpl per semplificare la creazione degli oggetti

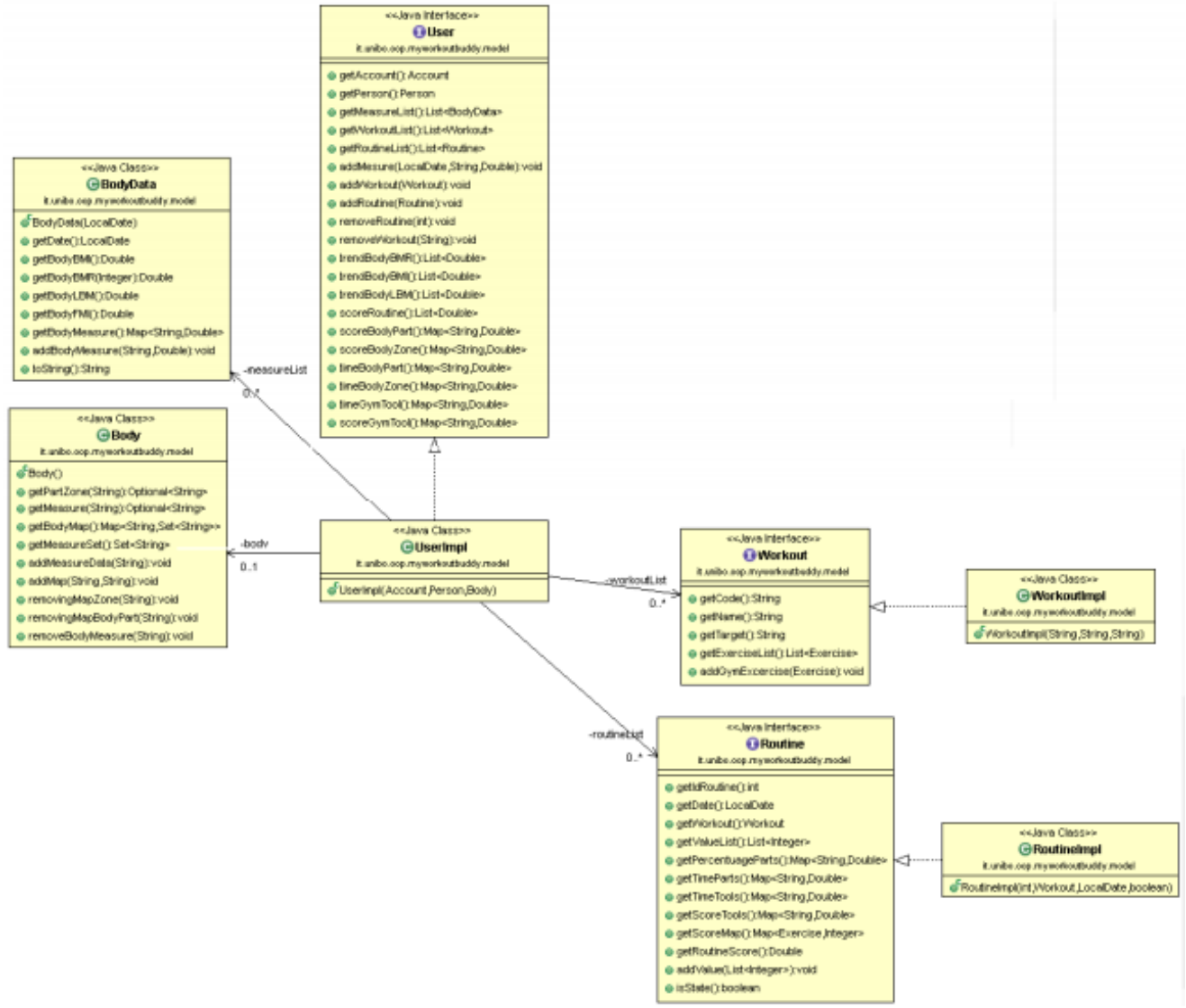
- Adapter: classi ManageUser, ManegeWorkout per adattamento al controller Uso di ArrayAsList per creazione di hashSet in MainJtestUnit

- Factory: Possibile utilizzo per definire body specifici per le diverse tipologie di utente (principiante, sportivo, potenziamento, mantenimento, ...). Il pattern non è stato realizzato per dare la completa scelta da parte dell'utente nella definizione del body.

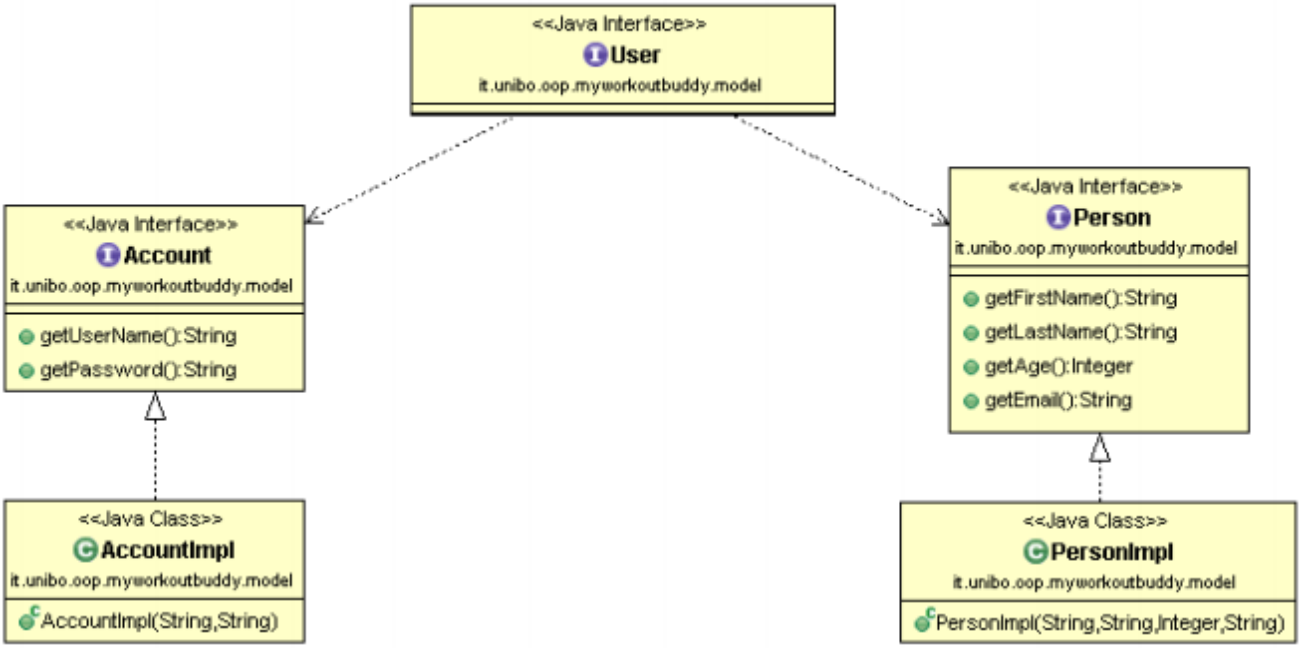
Model Schema di supporto



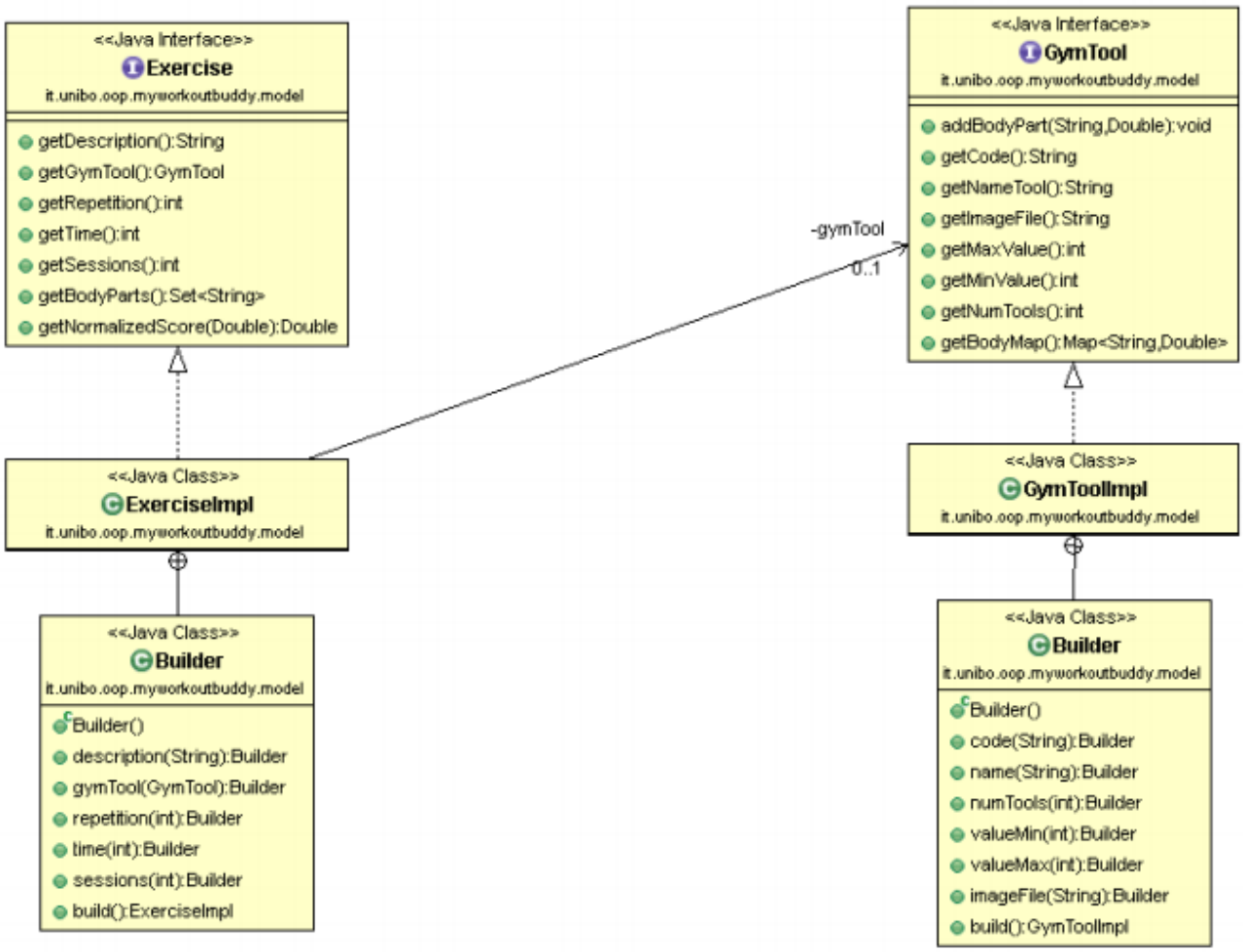
Model: Schema dettaglio User



Model: Schema dettaglio User - Account

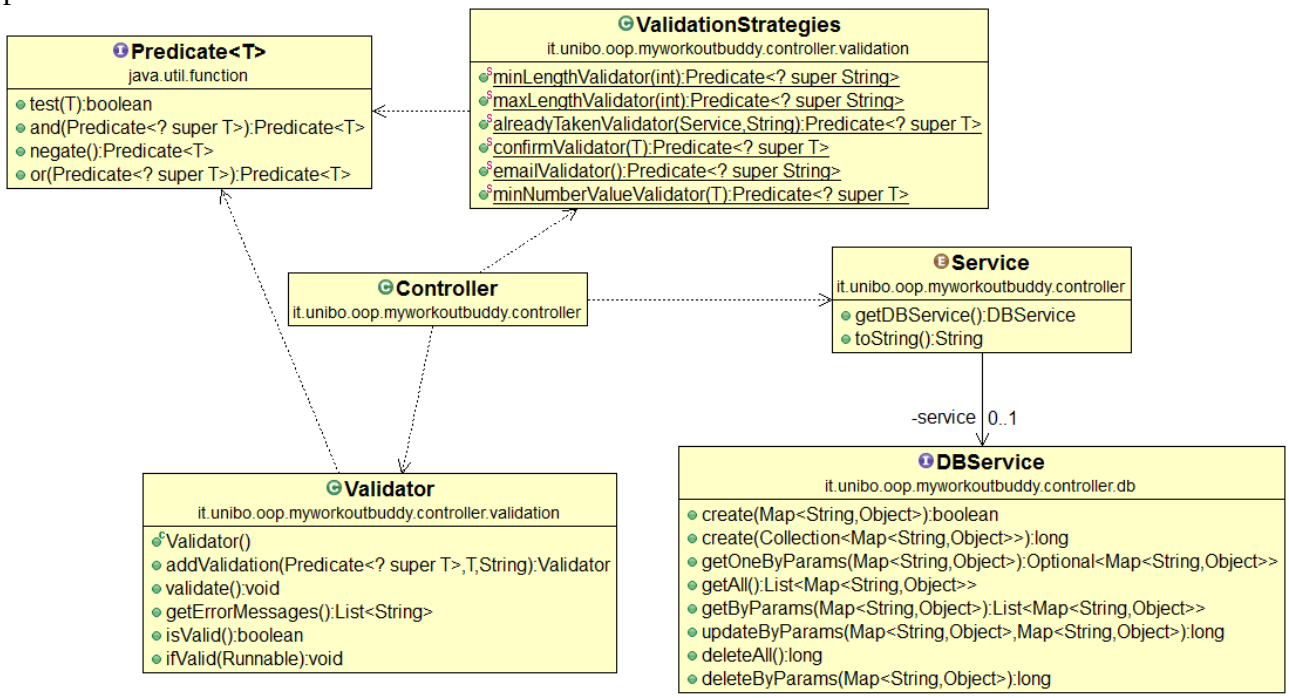


Model: Schema dettaglio Exercise e GymTool



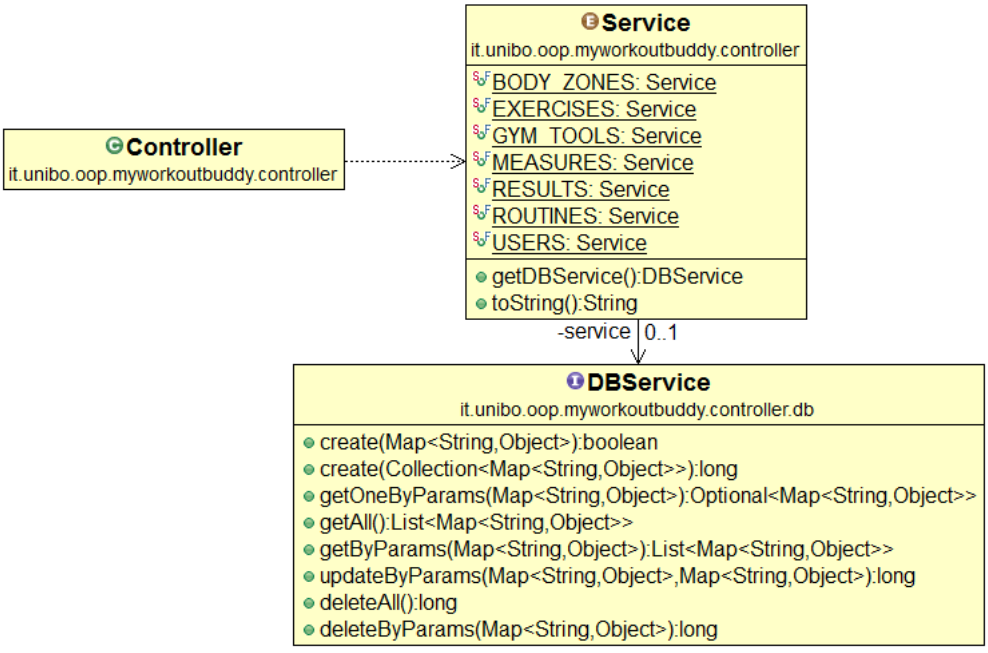
2.2.2 **Controller (Vandi Mattia)**

La radice della logica di controllo è la classe Controller che costituisce il collegamento tra il dominio applicativo e la logica di presentazione dell'applicazione. Il Controller si appoggia su un servizio per l'interazione con la base di dati scelta e su una strategia di validazione per il controllo dei dati provenienti dalla vista.



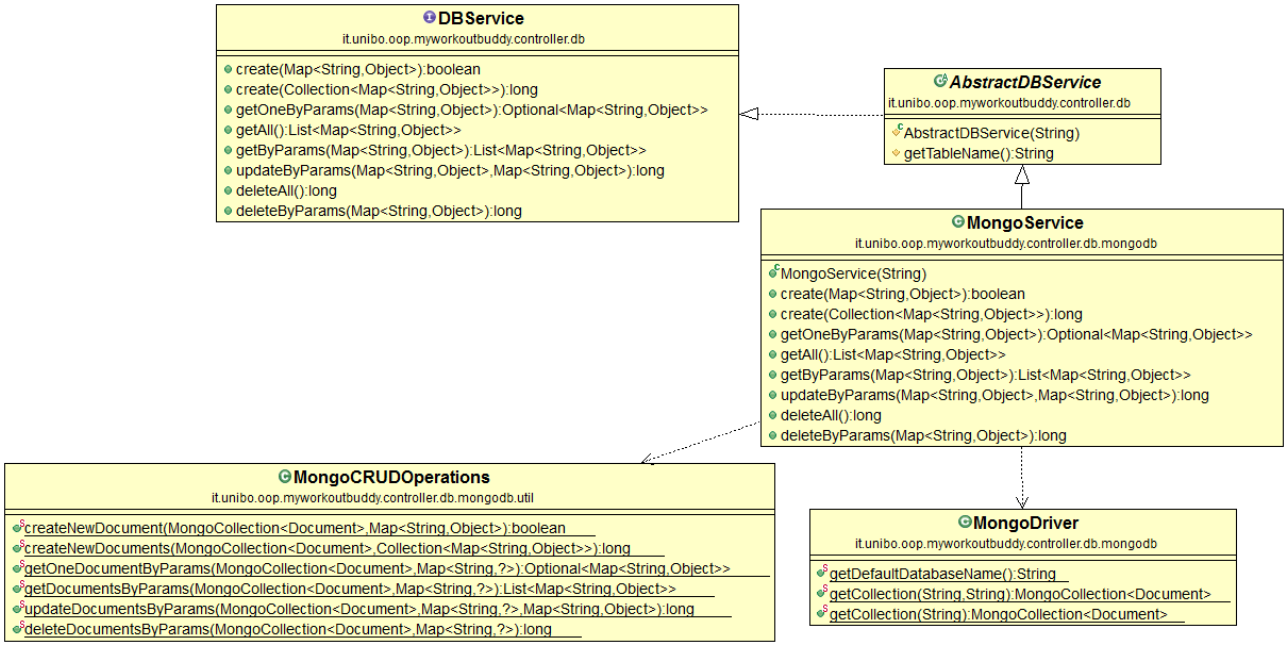
Schema di funzionamento del Controller.

Per rendere l'implementazione del Controller il più estendibile possibile, sono state prima individuate quali informazioni dovranno essere salvate permanentemente e sono state racchiuse in una enumerazione. In seguito è stato definito l'entry–point per l'interazione con la base di dati che si intende utilizzare.
Dopo aver definito tale entry–point ogni valore di tale enumerazione espone al Controller un metodo per richiamare un servizio che espone le operazioni da eseguire sulla di base di dati scelta.



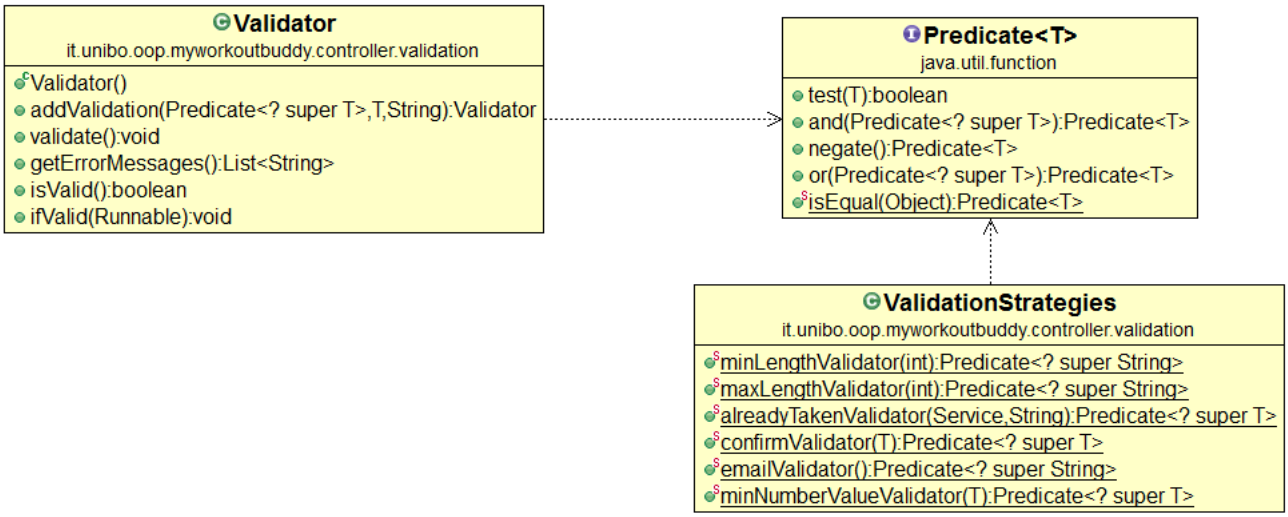
Schema di funzionamento generale per l'interazione con la base di dati

Dopo aver definito l'entry–point è stata definita una classe astratta (`AbstractDBService`) affinché tutte le implementazioni concrete abbiano una base di partenza comune. L'implementazione concreta (`MongoService`) si appoggia sulla classe `MongoDriver` per recuperare la connessione con il server e sulla classe `MongoCRUDOperations` per effettuare le operazioni sulla base di dati specifica.



Schema di funzionamento in dettaglio dell'interazione con la base di dati scelta.

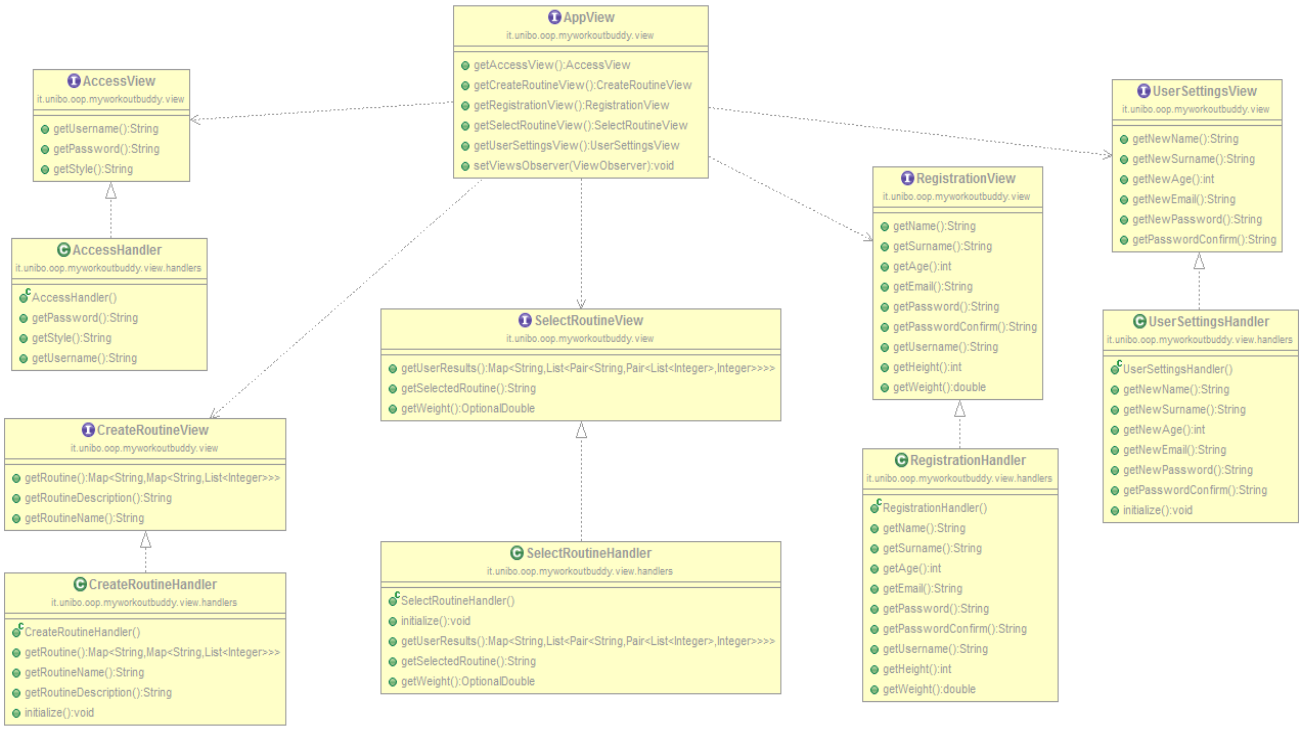
Dal momento che nella nostra applicazione un utente, prima di poterne sfruttare le funzionalità, deve essere registrato e avervi acceduto si è resa necessaria l'implementazione di una strategia per la validazione dei dati immessi sia in fase di registrazione che in fase di accesso.
Tale strategia si compone di un contesto (`Validator`) sul quale è possibile aggiungere strategie di validazione (`Predicate`) per tutti i valori per cui si ritiene necessario e associare a tale validazione un messaggio di errore in caso la validazione fallisca.
Tali messaggi di errore, uno per ogni valore che non ha passato la strategia di validazione, potranno essere recuperati una volta effettuata la validazione. Le strategie di validazione attualmente in uso sono state raggruppate in una classe di utilità.



Implementazione del pattern Strategy per la validazione dei dati immessi dall'utente. Tutte le strategie di validazione attualmente in uso sono rappresentate come metodi statici in una classe di utilità.

2.2.3 **View (Piscaglia Nicola)**

La radice dell'architettura di vista dell'applicazione è rappresentata dall'interfaccia AppView che costituisce per cui l'entry-point della View. Da questa interfaccia si può accedere alle varie funzionalità di ogni schermata costituite a loro volta da interfacce con metodi utili per recuperare dati dalla GUI. L'implementazione di ogni singola interfaccia risiede nelle rispettive classi Handler che implementano ognuna la corrispondente interfaccia.



Schema UML del View Design. App Views rappresenta l'entry-point della vista, le interfacce di view definiscono le funzionalità di ogni schermata di vista e sono implementate dai rispettivi handler.

Uno dei pattern più importanti, definiti fin dall'inizio, è l'Observer. La view è l'osservato dal controller che fornisce metodi per l'aggiornamento della GUI attraverso l'interfaccia ViewObserver. Questo ha permesso uno sviluppo parallelo delle funzionalità tra View e Controller e un maggior incapsulamento tra le parti, infatti la view non conosce l'implementazione dei metodi dell'osservatore. Quest'ultimi vengono chiamati dai vari Handler attraverso il metodo getObserver della classe ViewHandler che permette di reperire il riferimento al Controller e di utilizzarne le relative funzionalità.

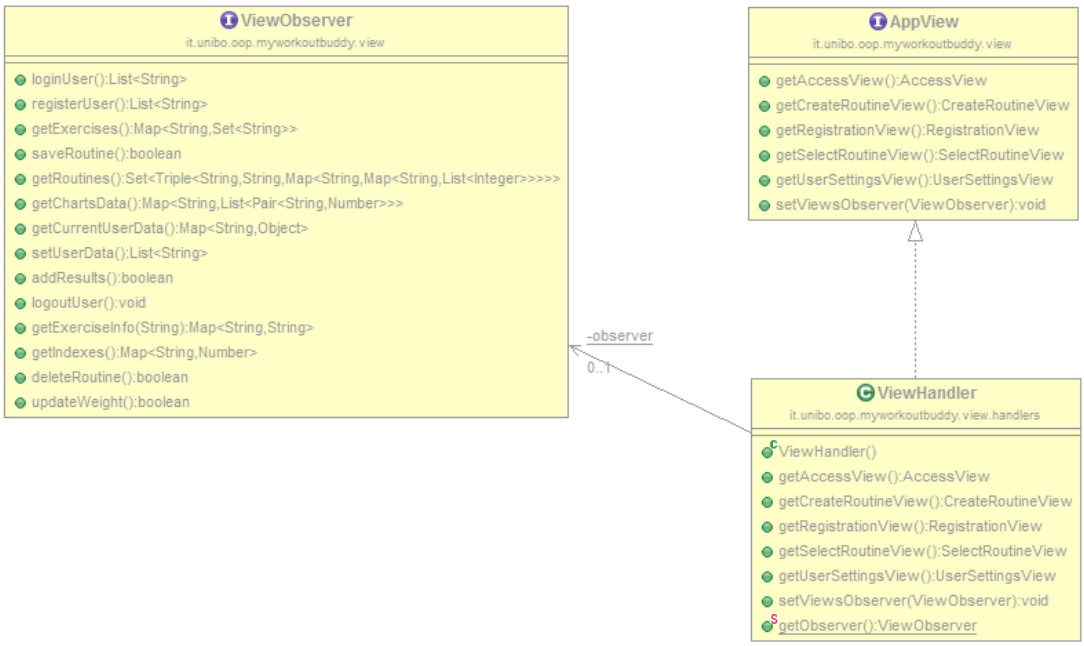
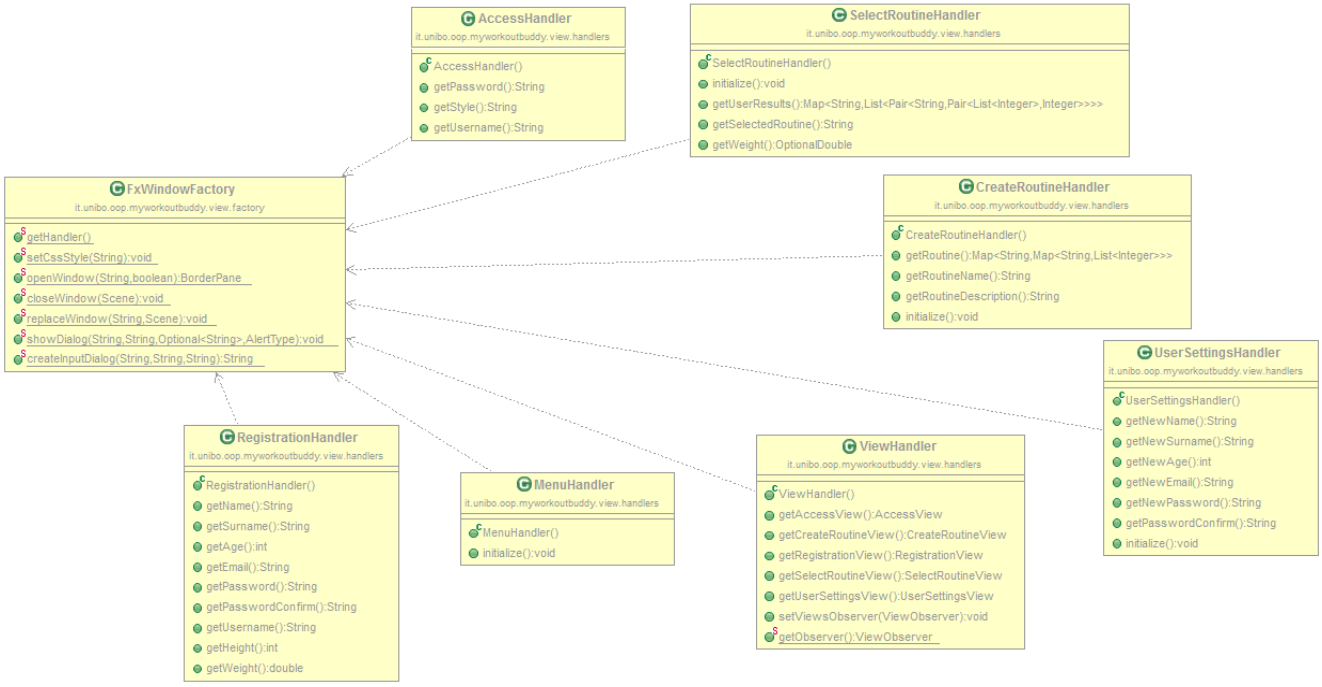


Figura 3. UML Pattern Observer. La classe ViewHandler, che implementa l'interfaccia AppView, si compone di un metodi per recuperare i riferimenti alle varie View ed anche all'osservatore (rappresentato dal Controller). I metodi dell'Observer sono definiti dall'interfaccia ViewObserver, di cui ViewHandler non conosce l'implementazione.

In merito ai pattern creazionali, sono stati utilizzati i pattern della Static e Simple Factory.
Per la costruzione della scena e delle finestre di dialogo con l'utente è stata implementata una classe FxWindowFactory contenente metodi statici necessari alla generazione delle varie view.
Generazione che viene effettuata da molteplici classi Handler come si può dedurre dallo schema UML sottostante, e che quindi ha permesso di fattorizzare in maniera importante codice semplificando l'implementazione degli Handler. La creazione di una StaticFactory ha permesso oltrettutto di incapsulare in tale classe, la gestione dello switch tra i vari stili di formattazione che l'applicazione fornisce.



Schema UML della Static Factory utilizzata. Si può notare come il suo uso sia stato impiegato nella maggioranza degli handler per permettere il caricamento di una nuova scena al susseguirsi di un evento.

In maniera simile è stato risolto il problema della creazione dei grafici utilizzando il pattern della Simple Factory. Per semplificare la costruzione di diversi tipi di grafici visualizzabili, essa è stata incapsulata in una classe separata chiamata `SimpleChartFactory` che implementa l'interfaccia `ChartFactory`. Il contesto della creazione di tali oggetti è la classe `StatisticsHandler`, dove viene inizializzata la GUI relativa alle statistiche, che contiene un oggetto di tipo `ChartFactory`.
Ogni metodo di tale interfaccia definisce un diverso tipo di grafico e restituisce il corrispondente riferimento.

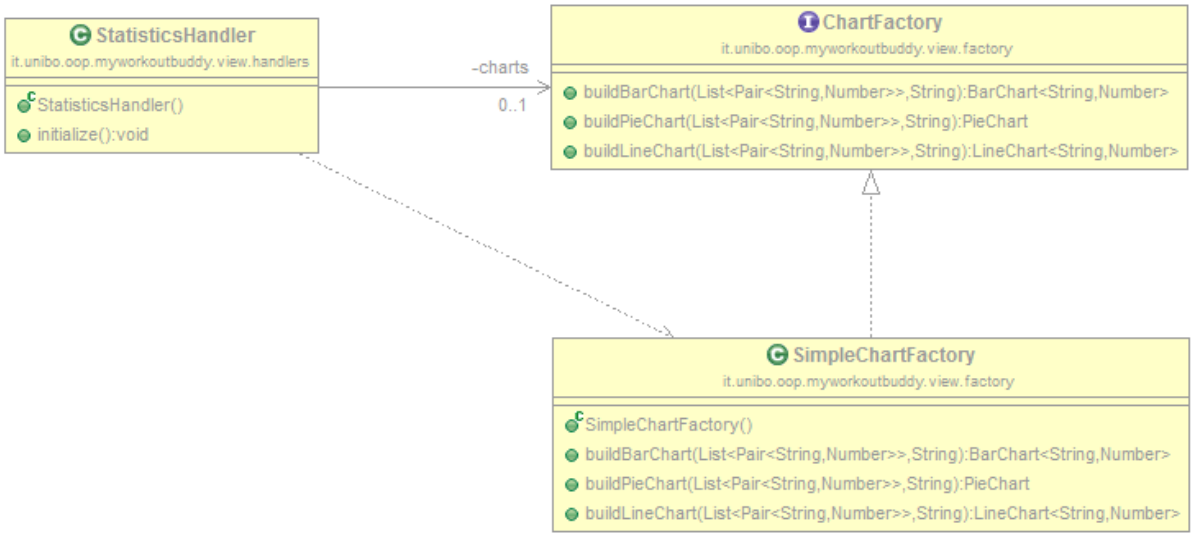


Figura 4. Schema della `SimpleFactory` *usata nella classe* `StatisticsHandler`*. Essa contiene un oggetto di Tipo* `ChartFactory` *che implementa metodi per la costruzione dei vari tipi di grafici richiesti.*

Nella mia progettazione ho trovato utile utilizzare anche pattern comportamentali quali lo Strategy. Nella gestione della vista di selezione delle schede di allenamento salvate, ho cercato di incapsulare la logica del layout di un allenamento in una classe `WorkoutLayout` separata che fornisce metodi per aggiungere un workout allo schema generale della scena e il rispettivo metodo per permettere di recuperarne di dati. Questo, a mio avviso, permette una più facile possibile modifica del layout di inserimento dei risultati della `SelectRoutineView` separando la parte statica di vista da quella dinamica di aggiunta dei risultati. Nel contesto della classe `SelectRoutineView`, infatti non vi è menzione della struttura scelta per l'implementazione grafica della GUI degli allenamenti.

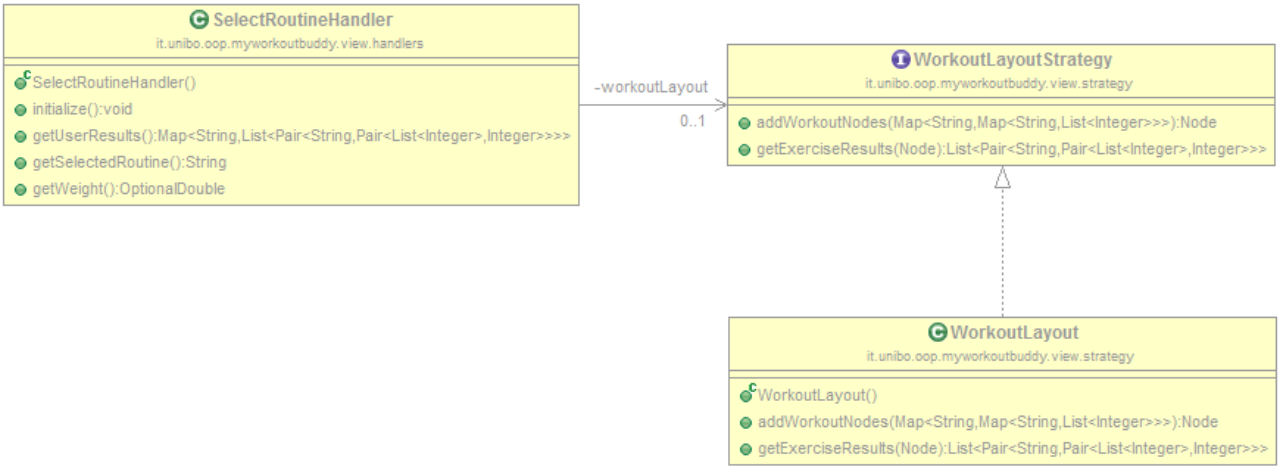
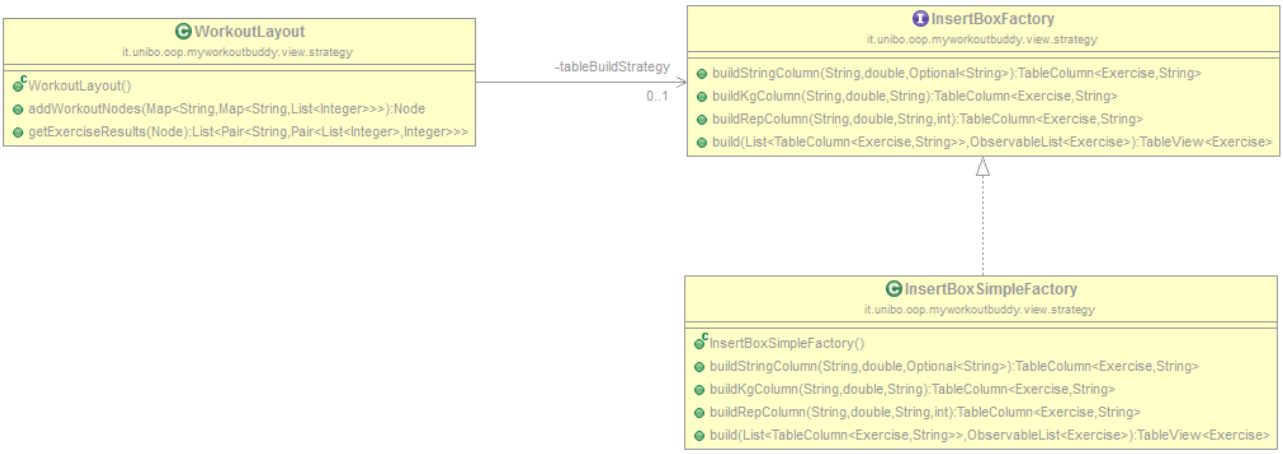


Figura 5. Schema del pattern Strategy utilizzato per il layout di ogni allenamento nella vista di selezione della Routine. Il layout di una allenamento viene specificato dai metodi dell'interfaccia WorkoutLayoutStrategy di cui si compone la classe SelectRoutine.

Nel contesto della strategia di layout di un allenamento ho ritenuto opportuno suddividere ulteriormente il problema della creazione di uno schema grafico, in un altro sorgente esterno, utlizzando una Simple Factory chiamata `InsertBoxSimpleFactory`, la quale implementa le funzionalità definite nell'interfaccia `InsertBoxFactory`. Ho scelto di utilizzare tale pattern poiché ho trovato di non banale realizzazione la costruzione del box di visualizzazione dei dati salvati e la gestione del loro inserimento. In tal modo la costruzione delle singole parti del box della `SelectRoutine` viene separata dalla parte di mapping dei dati da stampare nel box che vengono passati dal Controller. Questo permette anche di personalizzare facilmente la vista di ogni allenamento, inserendo diverse parti definite dall'interfaccia `InsertBoxFactory`.



Schema UML del Simple Factory usato nell'implementazione della struttura utilizzata per contenere i dati dell'allenamento. La classe `InsertBoxSImpleFactory` *incapusla i metodi per per aggiungere nuovi campi definiti dai metodi in* `InsertBoxFactory`.

L'ultimo pattern degno di nota è lo Strategy utilizzato nella classe `CreateRoutineHandler`. La strategia di controllo delle varie azioni eseguibili dall'utente interagendo con la GUI di creazione di una scheda, viene incapsulata in un oggetto separato, contenuto nell'Handler, che effettua i controlli e stampa a video finestre di dialogo con i relativi errori o informazioni. L'interfaccia implementata dalla classe `CreateRoutineCheck` definisce tutti i metodi per il check del corretto funzionamento della creazione della Routine, assicurando che l'utente non immetta nel sistema informazioni errate. Questa separazione ha permesso di semplificare in maniera significativa la classe `CreateRoutineHandler`, lasciando in essa solamente gli aspetti di gestione degli eventi e di recupero della routine creata.

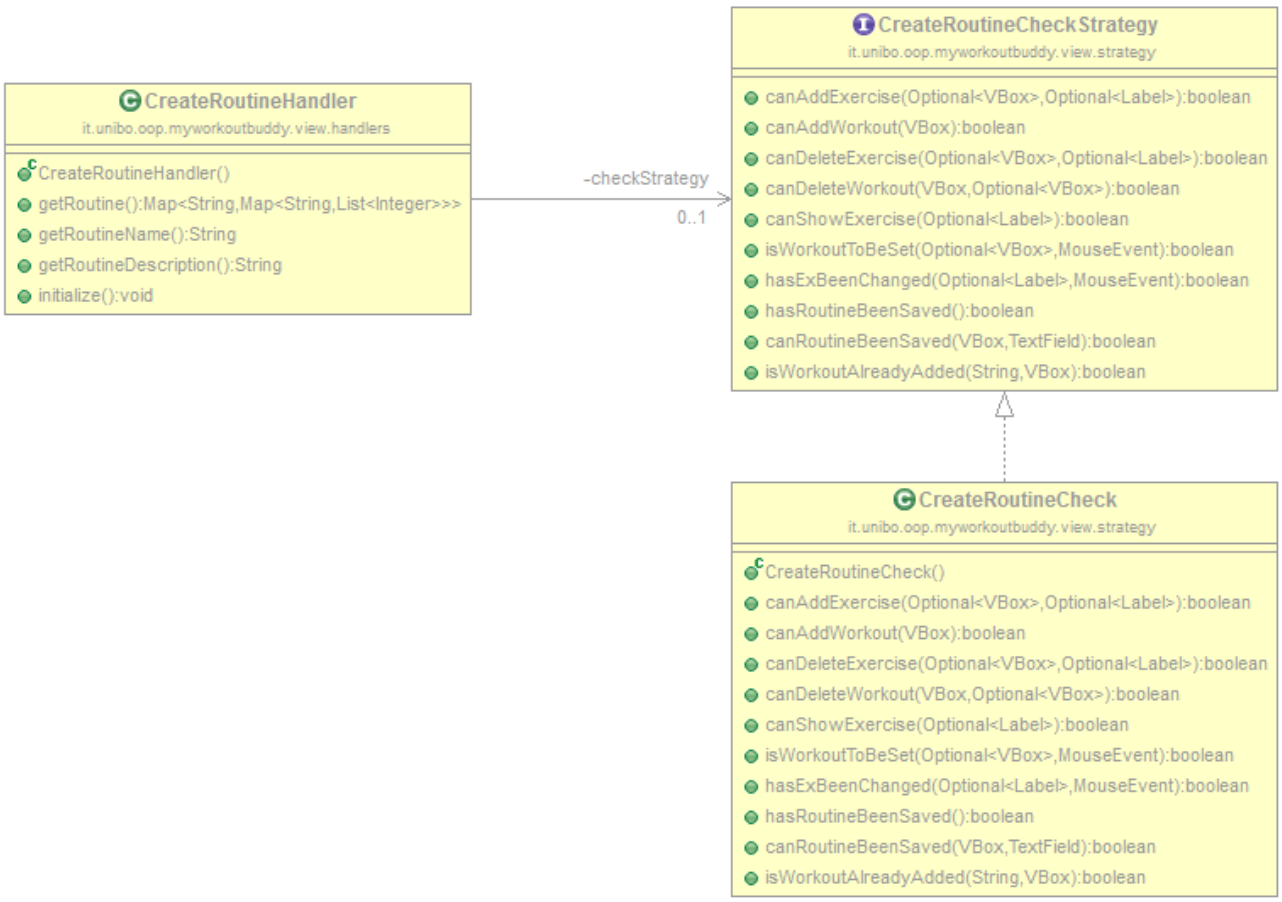


Figura 6. Schema UML del pattern Strategy utilizzato per il controllo sulla creazione di una routine. Un oggetto della classe CreateRoutineCheck viene istanziato nel contesto della CreateRoutineHandler e fornisce i metodi dichiarati nell'interfaccia Strategy.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

La creazione di una classe di funzionamento ideale del Model è stata realizzata nel package "model.test".
È stata testata la creazione di numerosi account con i corrispettivi utenti.
Fatto questo sono stati aggiunti all'utente corrente, ovvero l'utente che ha fatto l'accesso all' applicazione, misure delle proprie caratteristiche corporee, dati relativi a esercizi che ha voluto aggiungere precedentemente.
Infine sono state visualizzate in modo corretto, secondo il funzionamento del test, le statistiche dell'utente corrente e di tutti gli utenti, di quest'ultime solo quelle relative al tempo degli attrezzi da loro usati.
Si hanno due tipi di classi Test:
a) **MainJUnitTest**: Classe per il test automatizzato JUnit.
b) **MainTestModel** : Classe di prova del Model, con caricamento di dati e output di tutte le statistiche su console.
a) **MainJUnitTest**. Implementa un completo utilizzo del model con il caricamento dei dati ed una sequenza di allenamenti.

Si compone di due test: testLoadUsers e dataWorkoutUser.
Il testLoadUsers verifica il caricamento dei dati generali e degli utenti:
- Definizione di uno schema Body, con relativa verifica delle parti inserite
- Inserimento dei gymTool di esempio con verifica del numero e valori inseriti
- Definisce le percentuali di sollecitazione di ogni attrezzo sulle parti del corpo.

Verifica l'inserimento di dati errati (gymToolCode) che devono essere esclusi.
- Inserisce una lista di tre utenti con la verifica del numero e dell'utente corrente.

Il dataWorkoutUser() verifica i dati relativi all'attività di allenamento di un singolo utente e comprende:
- Inserimento di una serie di misurazioni iniziali del corpo, con verifica del calcolo del BMI e verifica della somma dei valori caricati
- Inserimento di una scheda allenamento (workout) con associata una lista di esercizi corretti e verifica della dimensione.
- Inserimento di esercizi errati e verifica dell'esclusione.
- Ciclo di inserimento delle sessioni di allenamento (routine) con data corrente e valori assegnati.
- Cancellazione dei dati di una specifica routine e verifica dimensione modificata.
- Inserimento misurazioni finali dell'utente e verifica nuovo calcolo BMI.
- Cancellazione del workout e verifica eliminazione di tutte le routine ad esso associate
- logout dell'utente e verifica sconnessione.

I dati delle statistiche sono verificati con il test MainTestModel e relative stampe.
MainTestModel. Esegue un ciclo di login e logout su tutti gli utenti caricati producendo l'output di tutte le statistiche e dei coefficienti calcolati.

Per quanto riguarda la parte del Controller, sono stati effettuati soprattutto test automatizzati per testare le funzionalità di base per l'interazione con il database. In questi test è stato verificato che i risultati a livello implementativo riflettevano la situazione presente sulla base di dati.

3.2 Metodologia di lavoro

Come si evince dal design di progetto, la suddivisione delle parti sono state effettuate nella seguente maniera:
- Pacini: Model
- Piscaglia: View
- Vandi: Controller + API Database MongoDB

Dopo una iniziale valutazione interna al gruppo, anche accogliendo il prezioso suggerimento del professore Viroli, abbiamo deciso di adottare la suddetta divisione per equilibrare al meglio il carico di lavoro di ciascun componente.

Nella progettazione del sistema le parti sono state sviluppate nella maniera più indipendente possibile ad eccezione delle sole interfacce entry-point di collegamento tra i vari componenti di design.
Ogni entry-point è stato definito in comune tra i due soli membri delle parti di competenza interposte tra i due collegamenti (model-controller e controller-view), il che ha permesso una maggiore autonomia implementativa.
L'approccio seguito è stato il più possibile di tipo *top–down* definendo prima le funzionalità di ogni singola parte definendo poi interfacce appropriate.

Il DVCS utilizzato è stato Mercurial con supporto di hosting BitBucket.
Lo sviluppo del programma si è limitato ad un solo branch di default con una release finale in un branch chiamato "stable".
Nell'utilizzo di tale strumento si è cercato di effettuare numerosi commit con messaggi significativi che potessero dare un'idea dei cambiamenti apportati a tutti i membri del gruppo.
Si è tenuto sotto controllo la tipologia dei file tracciati evitando di caricare sul repository file binari e altri file non indispensabili al progetto.

3.3 Note di sviluppo

In primis per la comunicazione col database sono state utilizzate le API fornite da MongoDB.
Per la realizzazione dell'interfaccia grafica, invece, è stata utilizzata la libreria grafica JavaFX che viene fornita come libreria stand-alone dal JDK 8. Nella costruzione della struttura statica della GUI sono stati utilizzati dei file FXML (assimilabili a i più comuni file XML) caricati dinamicamente run-time appoggiandosi alla classe FXMLLoader. Lo stile dell'interfaccia grafica è descritto, invece, in file CSS caricati anch'essi dinamicamente nella scena grafica.

Ove possibile abbiamo cercato di utilizzare librerie esterne affermate per rendere più efficace ed efficiente il nostro sviluppo.

Per la realizzazione del video player per la visualizzazione della corretta esecuzione degli esercizi è stato reperito in rete da una guida Oracle il codice della classe MediaControl.

Per quanto riguarda la validazione dell'input è stata utilizzata la libreria Apache Commons Validator per controllare la correttezza delle email digitate dell'utilizzatore finale. Invece per la realizzazione della classe Validator si è reso necessario l'utilizzo di una tabella per garantire l'unicità tra il valore da controllare e il corrispondente algoritmo di validazione. Tale tabella è stata reperita dalla libreria Google Guava.

Per lo scambio di dati tra controller e view si è reso necessario un uso estensivo delle classi Pair e Triple fornite dalla libreria Apache Commons Lang3.
Ad ogni operazione effettuata dall'utente il Controller effettua il controllo dello stato dell'applicazione (e.g. un utente non ha ancora effettuato l'accesso ma richiede di poter visualizzare

la sua scheda di allenamento) è stata utilizzata l'utility-class Preconditions della libreria Google Guava.

Di seguito si elencano le formule di calcolo dei coefficienti:

- BMI (Body Mass Index) BMI = Weight / (Height * Height) Weight = Peso in kg Height = Altezza in m;
- BMR (Basal Metabolic Rate) Maschi: BMR = 5.0 + (10.0*Weight + 6.25*Height) – 5.0 Age Weight = Peso in kg Height = Altezza in cm Non viene gestito il calcolo per gli utenti di sesso femminile;
- LBM (Lean Body Mass) formula James : LBM = (1.10 * Weight) - 128 * ((Weight) 2 / (Height)2) Weight = Peso in kg Height = Altezza in cm.

Capitolo 4

Commenti finali

Il lavoro e l'impegno dedicato alla progettazione di questa applicazione, a nostro avviso, ha portato ad un buon risultato. Tra i punti di forza dell'applicazione sicuramente risalta la possibilità di personalizzazione del proprio allenamento, la portabilità dei dati degli utenti fornita dalla memorizzazione in una base di dati, un interfaccia grafica ben incapsulata nelle sue parti di stile, struttura, presentazione e gestione degli eventi anche grazie alla libreria Java FX. A questo proposito penso, per quanto riguarda la parte di View, di aver prodotto una buona GUI sia a livello grafico sia a livello di qualità del codice, cimentandomi con una nuova libreria grafica anche se avrei potuto dare una architettura più definita alle varie classi.
La memorizzazione online dei dati dell'utente rappresenta un vantaggio in termini di disponibilità dei dati, i quali possono essere reperiti da qualsiasi piattaforma avente l'applicazione, anche se presenta due punti di debolezza perché vincola il reperimento dei dati alla disponibilità di rete internet e lega la velocità di recupero alla banda disponibile sulla linea e alla connessione al server in cui i dati sono salvati.
Nonostante alcune feature opzionali non siano state implementate, l'applicazione ben si presta a successivi miglioramenti e sviluppi successivi in particolar modo per quanto riguarda la produzione di grafici, statistiche e consigli per l'utente, anche con l'integrazione di conoscenze più specifiche di Scienze Motorie.

Appendice A

Guida utente

Per iniziare ad utilizzare l'applicazione è necessario registrare un account ed autenticarsi utilizzando le apposite schermate disponibili all'avvio dell'applicazione. In alternativa ad una nuova registrazione, può essere utilizzato l'account di prova con dati: Username = "marcorossi", Password = "pass00". Ricordiamo che il processo di autenticazione e altre operazioni di modifica di dati possono essere soggetti a modesti ritardi di risposta dovuti al tempo di interazione con il database MongoDB. Una volta effettuata l'autenticazione per accedere alle diverse funzionalità del sistema, è sufficiente selezionare le voci del menù in vista sulla sinistra. L'utilizzo delle varie schermate è relativamente semplice e non merita un prolisso approfondimento.

Bibliografia

Codice per la classe `MediaControl`:
https://docs.oracle.com/javafx/2/media/playercontrol.htm/

Google Guava:
http://github.com/google/guava/

Apache Commons Lang3:
http://commons.apache.org/proper/commons-lang/

Apache Commons Validator:
http://commons.apache.org/proper/commons-validator/