Qontainer

Progetto di Programmazione ad Oggetti
A.A. 2018/2019

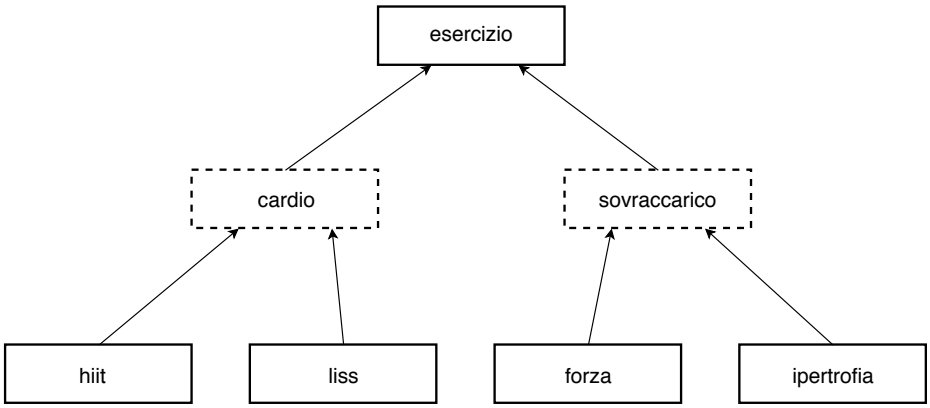
Nicoletta Fabro 1143541

1. Descrizione generale

`Qontainer` è un contenitore che si occupa di gestire una gerarchia di esercizi fitness. Tale contenitore vuole fornire un modo pratico e veloce di tenere traccia degli esercizi eseguiti durante le proprie sessioni di allenamento registrandone nome, data ed altre informazioni che variano a seconda del tipo di esercizio inserito.

2. Descrizione delle classi

La gerarchia di classi usata per modellare la realtà in questione è la seguente:



`esercizio`

Ereditarietà

E' la classe base della gerarchia. Per la realtà in questione non è stato ritenuto opportuno rendere tale classe base astratta, in quanto i metodi virtuali puri che si incontrano successivamente nella gerarchia sono tipici delle due sottoclassi `cardio` e `sovraccarico`.
Il distruttore di esercizio è virtuale.

Attributi

`nome`: è un tipo stringa che rappresenta il nome dell'esercizio.
`dataEsecuzione`: per rappresentare correttamente la data di esecuzione dell'esercizio (e, come si vedrà più avanti nel caso di alcuni esercizi, la durata) sono state definite due classi *ad hoc* `data` e `orario`, definite nei file `classidiutilita.h` e `classidiutilita.cpp`. Queste due classi sono state sviluppate basandosi sugli esercizi visti durante il corso ed, inoltre, sono stati forniti per ciascuna classe due metodi di conversione dal tipo di dato al formato stringa e viceversa.

Metodi polimorfi

I metodi dichiarati `virtual` in `esercizio` sono gli operatori di confronto == e `!`= ed il metodo `serializza(char)`, usato dalla classe `database` per salvare l'istanza della classe su file.

Metodi importanti
In `esercizio` sono presenti gli usuali metodi di utilità per accedere agli attributi della classe o modificarli, quali `getNome()`, `getDataEsecuzione()`, `setNome()` e `setDataEsecuzione()`.

`cardio`

Ereditarietà
Deriva pubblicamente da `esercizio` ed è una classe astratta. Indica tutti gli esercizi che vanno ad allenare la capacità cardiovascolare dell'organismo.
Attributi
`bpm`: sta per *battiti per minuto* ed indica la frequenza cardiaca (FC) dell'individuo durante l'allenamento. E' rappresentato come `int`.
`eta`: età dell'individuo, rappresentata come `int`, serve per calcolare la FC massima che verrà poi utilizzata per ricavare le diverse zone allenanti utilizzando la formula di Karvonen.
Metodi polimorfi
`durataAllenamento()`: è un metodo virtuale puro, in quanto la durata dell'allenamento viene calcolata in modo diverso a seconda che si abbia a che fare con un allenamento a circuito (hiit) o con un allenamento "continuativo" (liss). Sono inoltre virtuali gli operatori di confronto ed il metodo `serializza(char)` usato per scrivere i dati su file.
Metodi importanti
Oltre ai metodi di utilità per accedere o modificare gli attributi della classe, di particolare rilevanza sono i metodi `FCmassima()` e `percentualeFC()`, che calcolano rispettivamente la frequenza cardiaca massima dell'individuo (secondo una formula standard) e la zona allenante in cui il soggetto si trova durante l'allenamento (espressa in percentuale e calcolata usando la formula di Karvonen). Entrambi i metodi ritornano un valore intero.

`sovraccarico`

Ereditarietà
Deriva pubblicamente da `esercizio` ed è anch'essa una classe astratta in quanto contiene il metodo virtuale puro `volume()`. Indica tutti gli esercizi per i quali serve l'ausilio di un sovraccarico (inteso anche come peso corporeo) e che vanno ad allenare la capacità del muscolo di rispondere ad uno stimolo del sistema nervoso centrale (forza) o ad aumentare di diametro (ipertrofia).
Attributi
`serie`: è un intero che indica il numero di serie effettuate del medesimo esercizio.
`ripetizioni`: è un intero che indica il numero di ripetizioni effettuate per ciascuna serie di quell'esercizio. E' stato deciso di non dare la possibilità all'utente di variare il numero di ripetizioni all'interno delle serie per un fattore propriamente progettuale (il numero di ripetizioni inserito resta uguale per tutte le serie di quell'esercizio), tuttavia ciò costituirebbe uno sviluppo futuro interessante al fine di implementare metodiche di allenamento differenti, quali ad esempio l'allenamento piramidale.
Metodi polimorfi
`volume()`: il volume dell'allenamento è un numero che si usa per quantificare il lavoro svolto e si ottiene con la formula `serie*ripetizioni(*altri fattori opzionali)`. E' calcolato diversamente a seconda che si parli di forza o di ipertrofia e per questo è stato marcato virtuale puro. Sono inoltre virtuali gli operatori di confronto ed il metodo `serializza(char)` usato per scrivere i dati su file.
Metodi importanti
A parte i metodi di utilità per accedere o modificare gli attributi della classe, in `sovraccarico` non sono presenti altri metodi caratteristici della classe.

`hiit (high intensity interval training)`

Ereditarietà
Deriva pubblicamente da `cardio`. Indica tutti gli esercizi ad alta intensità, dove la frequenza cardiaca si aggira attorno all'85%-95% della propria FC massima. Vista l'alta intensità a cui sono eseguiti solitamente questi esercizi si presentano sotto forma di allenamenti a circuito, costituiti cioè da brevi intervalli di lavoro ad alta intensità seguiti da brevi intervalli di recupero attivo (esempi di protocolli di allenamento ad alta intensità sono l'allenamento Tabata o le ripetute).
Attributi
`numeroIntervalli`: è un intero che indica il numero di intervalli di lavoro compiuti durante l'allenamento.

durataIntervalli: è un orario (definito nei file classidiutilita.h e .cpp) che indica la durata degli intervalli di lavoro.
riposo: è anch'esso un orario che indica la durata degli intervalli di recupero attivo tra un intervallo di lavoro e l'altro.

Metodi polimorfi

La classe hiit implementa il metodo virtuale puro durataAllenamento() definito in cardio. Sono inoltre virtuali gli operatori di confronto ed il metodo serializza(char) usato per scrivere i dati su file.

Metodi importanti

In hiit è presente il metodo statico deserializza(const vector<string>&) che, dato un vettore di stringhe (corrispondente al dato salvato su file), ritorna un puntatore a hiit che punta al dato caricato da file.

liss (low intensity steady state training)

Ereditarietà

Deriva pubblicamente da cardio. Indica tutti gli esercizi cardio la cui intensità è tale da essere praticati anche per tempi medio-lunghi (corrispondente al 60%-80% della frequenza cardiaca massima del soggetto).

Attributi

durata: è un orario che indica la durata dell'allenamento.

Metodi polimorfi

La classe liss implementa il metodo virtuale puro durataAllenamento() definito in cardio. Sono inoltre virtuali gli operatori di confronto ed il metodo serializza(char) usato per scrivere i dati su file.

Metodi importanti

Anche in liss è presente il metodo statico deserializza(const vector<string>&) che ritorna un puntatore a liss che punta al dato caricato da file.

forza

Ereditarietà

Deriva pubblicamente da sovraccarico. Indica tutti gli esercizi che vanno ad allenare la capacità di risposta delle fibre muscolari ad uno stimolo del sistema nervoso centrale (CNS). In un protocollo di forza le ripetizioni sono basse (1-5) ed il numero di serie è più alto rispetto ad un protocollo di ipertrofia (nella forza le serie sono solitamente>=5).

Attributi

peso: è un intero che indica il carico sollevato in kilogrammi. Conoscere il carico sollevato è fondamentale negli allenamenti di forza per calcolare il proprio massimale (cioè il peso con cui si può effettuare 1 ripetizione, detta anche 1RM) e per calcolare la percentuale sollevata rispetto alla propria 1RM.

Metodi polimorfi

Il metodo volume() viene implementato in forza moltiplicando le serie per le ripetizioni per il carico sollevato.

Metodi importanti

A parte i metodi di utilità per accedere o modificare gli attributi della classe, in forza sono degni di nomina i metodi massimale() e percentualeCarico(): massimale() calcola la 1RM dato il peso correntemente sollevato, le serie e le ripetizioni. Siccome le formule per il calcolo del massimale usano numeri decimali, il calcolo è stato prima fatto in decimale e poi ritornato sotto forma di intero. Il metodo percentualeFC() invece calcola la % di peso che si sta sollevando dato il carico inserito rispetto al proprio massimale. Si ha inoltre il metodo statico deserializza(const vector<string>&) che ritorna un puntatore a forza quando si carica un dato da file.

ipertrofia

Ereditarietà

Deriva pubblicamente da sovraccarico. Indica tutti gli esercizi il cui scopo è quello di aumentare il diametro della fibra muscolare. In un protocollo di ipertrofia le ripetizioni sono superiori alle 6 e inferiori alle 15 e un fattore determinante è dato dall'intensità, ovvero dal tempo di recupero tra una serie e l'altra (minore è il tempo di recupero e maggiore è l'intensità).

Attributi

recupero: è un attributo di tipo orario che indica il tempo di riposo tra una serie e l'altra.

Metodi polimorfi
Come in `forza`, anche in `ipertrofia` viene implementato il metodo virtuale `volume()`. Nel caso dell'ipertrofia il volume dell'allenamento si ottiene moltiplicando il numero di serie per le ripetizioni per l'intensità dell'allenamento (calcolata come spiegato in "Metodi importanti").
Metodi importanti
Oltre ai metodi di utilità per accedere o modificare gli attributi della classe, in `ipertrofia` è presente il metodo `intensita()`, che ritorna un intero che serve a quantificare l'intensità dell'allenamento, che viene poi usato nel calcolo del volume. E' inoltre presente il metodo statico `deserializza(const vector<string>&)` che ritorna un puntatore a i`pertrofia` quando si carica un dato da file.

`Container<T>`
`Container<T>` è stato implementato sotto forma di *doubly linked list* in quanto nel caso di eliminazione di un elemento in posizione arbitraria è più efficiente, ovvero è eseguita in tempo costante. Per essere implementato sotto forma di lista doppiamente linkata `Container<T>` fa uso di altre tre classi annidate, quali:
1. `Nodo`: è la componente strutturale della lista doppiamente linkata. Ogni nodo contiene un campo `info` di tipo `T` e due campi puntatore `prev` e `next`, che puntano rispettivamente al nodo precedente e a quello successivo;
2. `Iterator`: iteratore usato per scorrere la lista in lettura e in scrittura, implementato seguendo i principi visti in aula;
3. `const_Iterator`: iteratore costante usato per scorrere la lista in lettura, implementato seguendo i principi visti in aula.

Vista l'assenza di operazioni che copiano la lista è stato deciso di non implementare un eventuale `DeepPtr<T>`.

3. Descrizione delle chiamate polimorfe
Si hanno delle chiamate polimorfe nei seguenti casi:
- Nella serializzazione, il metodo `std::string serializza(char)`, dichiarato virtuale in `esercizio`, viene invocato dal metodo `save(Container<esercizio*>&)`all'interno della classe `database` per salvare i dati su file;
- Nell'interfaccia grafica, nella pagina di gestione degli esercizi, cliccando sul bottone "Elimina" di un dato esercizio viene invocato il metodo `void remove(const T)` definito in `Container<T>`, il quale richiama l'operatore di confronto ridefinito virtuale `bool operator==(const T&)` già visto precedentemente;
- Sempre nell'interfaccia grafica, anche nel caso della ricerca di un esercizio, nell'invocazione del metodo `std::vector<T> search(const T val, typename std::enable_if<std::is_pointer<T>::value>::type* =0)` viene implementato l'operatore di confronto `operator==(const T&)`. Per il metodo `search` ho deciso di sfruttare due *feature* di C++11 quali `enable_if` e `is_pointer` per rendere la funzione di ricerca efficace sia nel caso T sia di tipo puntatore che non;
- Nella visualizzazione di un esercizio nell'interfaccia grafica, a seconda della categoria di appartenenza di questo, vengono invocate alcune delle funzioni polimorfe viste sopra: ad esempio, se si visualizza un esercizio di tipo `forza` vengono invocati i metodi `massimale()` e `percentualeCarico()`, mentre se visualizzo un esercizio di tipo `ipertrofia` viene invocato il metodo `volume()`. Analogamente, alla visualizzazione di un esercizio di tipo `hiit` o `liss` viene invocato l'opportuno override di `durataAllenamento()`.

4. Compilazione ed esecuzione
La compilazione si esegue tramite i comandi `qmake` ed in seguito `make`.

5. Caricamento/salvataggio dei dati

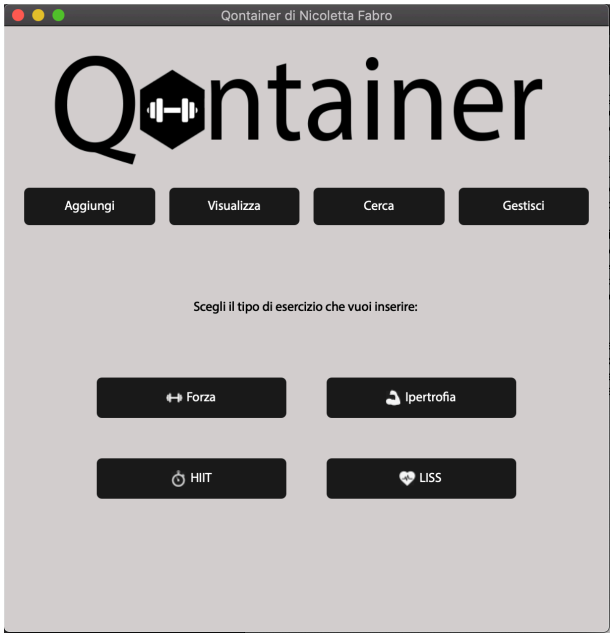
All'apertura della GUI, all'utente viene chiesto di scegliere il file da caricare (che si trova nella *root folder* del progetto, `Qontainer`, sotto il nome di `db.txt`).
E' stato scelto di definire una classe dedicata `database` la quale, mediante i metodi `load(Container<esercizio*>&)` e `save(Container<esercizio*>&)`, si occupa di caricare dati da un file di tipo `.txt` e di salvarli sul medesimo file. Tale classe invoca i metodi `serializza(char)` e `deserializza(const vector<string>&)` visti prima. I metodi `load` e `save` sono invocati nella GUI rispettivamente all'apertura (invocazione del costruttore della main window) e alla chiusura di questa (mediante l'overriding del metodo QCloseEvent, come da documentazione Qt). E' presente inoltre una funzione di controllo `controllo()` che controlla che venga selezionato il file `.txt` con il giusto contenuto (se non viene scelto il file giusto, viene richiesto di scegliere un altro file).

6. Interfaccia grafica

L'interfaccia grafica vuole essere il più chiara ed essenziale possibile. Per questo è stato deciso di implementare un menu orizzontale con le quattro azioni principali fornite da `Qontainer`: **Aggiungi**, **Visualizza**, **Cerca** e **Gestisci** (che comprende **Modifica** ed **Elimina**). Tale menu resta sempre visibile a prescindere dalla pagina in cui ci si trova e l'utente può sempre farci riferimento.

6.1. Aggiungi

All'apertura di `Qontainer,` dopo aver selezionato il file `.txt` da caricare, la schermata visualizzata è quella mostrata a sinistra. Sotto al logo e al menu appare di default la pagina per aggiungere un nuovo esercizio (raggiungibile anche cliccando **Aggiungi**). Cliccando sulla categoria dell'esercizio che si vuole aggiungere si viene riportati alla schermata di aggiunta avanzata (schermata di destra), dove verrà chiesto di riempire gli appositi campi per la tipologia di esercizio che si ha deciso di inserire.



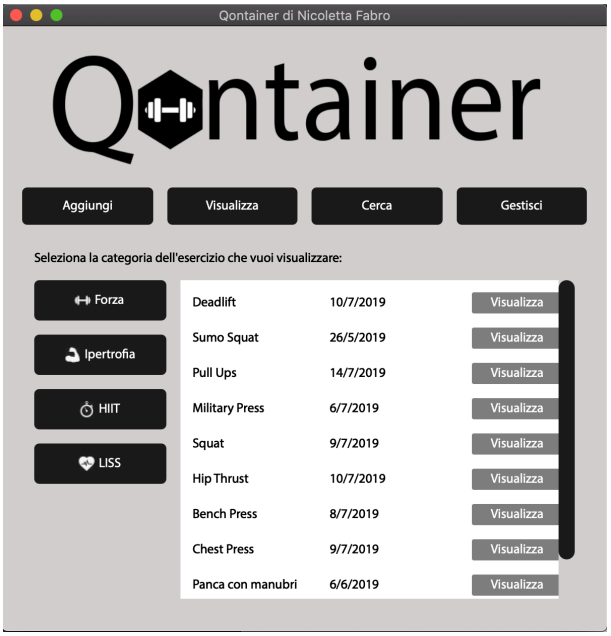
Oltre al menu prima menzionato, si può notare come sia presente un'ulteriore possibilità di scegliere la categoria dell'esercizio da inserire: viene così data la possibilità all'utente di cambiare idea senza dover cliccare nuovamente su **Aggiungi**. A seconda della categoria selezionata, i *fieldset* da riempire cambiano per corrispondere a quelli richiesti per creare l'esercizio in questione.

IMPORTANTE: come specificato dai *placeholder* dei campi da aggiungere, quando si inseriscono la data di esecuzione e la durata di un esercizio è importante non far precedere i numeri ad una cifra da 0, in quanto tale formato non è supportato e ciò risulterà in una data/durata uguale a 0.

Tutti i bottoni **Annulla** presenti in `Qontainer` riportano sempre alla *Home page*.

6.2. Visualizza

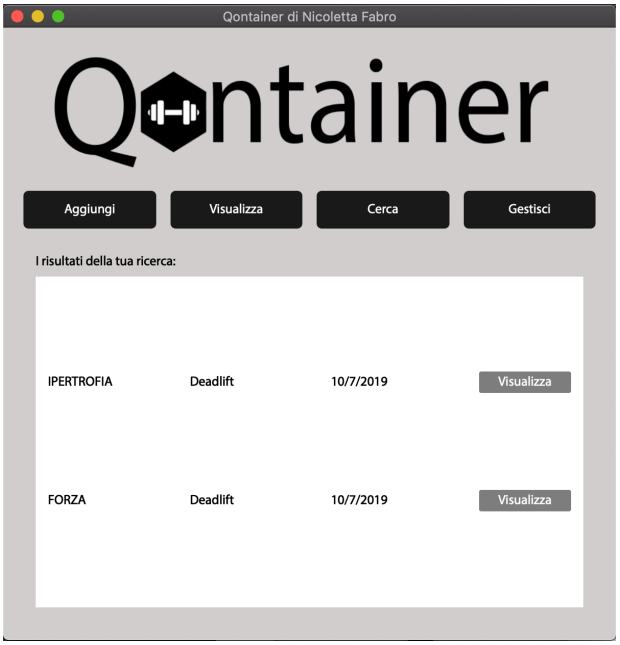
Cliccando su **Visualizza** viene visualizzata la schermata di sinistra:



L'utente viene invitato a scegliere la categoria di cui vuole visualizzare gli esercizi. A seguito di ciò compariranno gli esercizi presenti in quella categoria (come nell'immagine di destra).
Ogni singolo esercizio ha associato un bottone **Visualizza**, che porta alla schermata di visualizzazione del singolo esercizio.

6.3. Cerca

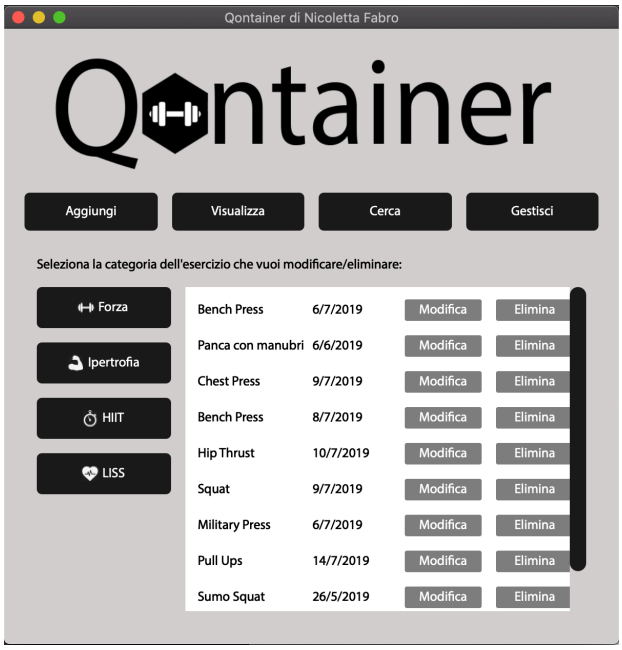
La sezione **Cerca** si presenta come segue:

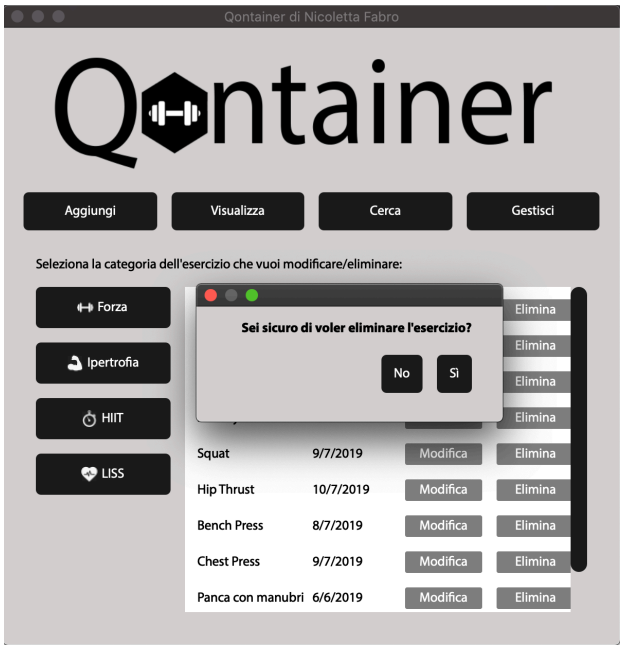


E’ stato deciso di implementare la ricerca solo a livello di nome e di data di esecuzione in quanto sono due campi facilmente memorizzabili (dover cercare un esercizio inserendo tutti i campi necessari è un operazione lunga e non sempre l’utente si ricorda esattamente cosa cerca).
Il risultato della ricerca -se questa va a buon fine- presenta gli esercizi con corrispondente nome e data di esecuzione *a prescindere* dalla categoria a cui questi appartengono (per vedere una ricerca che produce più elementi, cercare **Deadlift** con data **2/7/2019**). La categoria di appartenenza dell’esercizio nel risultato della ricerca viene scritta in stampatello vicino al nome dell’esercizio come mostrato di sopra (immagine di destra).

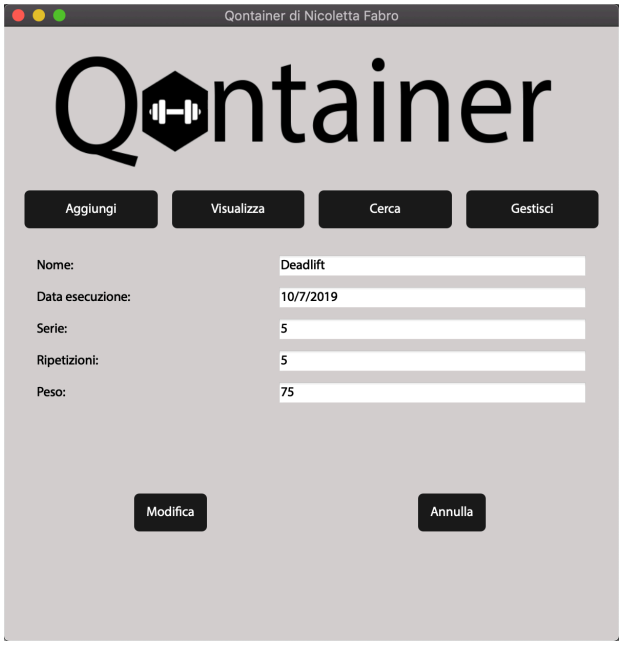
6.4.Gestisci

La pagina **Gestisci** racchiude le azioni di **Modifica** ed **Eliminazione** di un esercizio. Strutturalmente si presenta come la pagina di visualizzazione, con la differenza delle azioni associate ai singoli esercizi.





Quando si decide di modificare un esercizio si viene riportati alla pagina di modifica, mentre quando si clicca su **Elimina** viene chiesta conferma mediante un messaggio e, se si clicca su **Sì**, l’esercizio viene eliminato (immagine sopra, schermata di destra).



Nella pagina di **Modifica** è possibile modificare tutti i campi riguardanti l’esercizio selezionato.

Come già sottolineato precedentemente, tutti i bottoni **Annulla** riportano alla *Home page*.

Alternativamente l’utente può muoversi navigando con il menu orizzontale presente nella zona superiore della finestra.

7. Tempo richiesto

Analisi preliminare del problema:	1h
Progettazione del modello:	2h
Progettazione della GUI:	2h
Apprendimento libreria Qt:	10h
Codifica modello e GUI:	25h
Debugging:	5h
Testing:	5h

8. Specifiche

Sistema Operativo di Sviluppo: Mac OS Mojave 10.14.5
Compilatore: Apple LLVM version 10.0.1 (clang-1001.0.46.4)
Versione Qt: Qt 5.12.2
Versione Qt Creator: 4.9.0