

# Software per la gestione di una palestra

Tommaso Botarelli

Elaborato di ingegneria del software



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Motivazione . . . . .	3
1.2	Statement . . . . .	3
1.3	Estensioni future . . . . .	4
<b>2</b>	<b>Analisi e progettazione</b>	<b>5</b>
2.1	Use Case Diagram . . . . .	5
2.2	Use Case Templates . . . . .	8
2.3	Mock ups . . . . .	12
2.3.1	LoginPage . . . . .	12
2.3.2	Cliente . . . . .	12
2.3.3	Receptionist . . . . .	14
2.3.4	Personal trainer . . . . .	15
2.3.5	Varie . . . . .	16
2.4	Page navigation diagram . . . . .	17
2.5	Class diagram . . . . .	18
2.6	Pattern utilizzati . . . . .	19
2.6.1	Abstract factory . . . . .	19
2.6.2	Singleton . . . . .	20
<b>3</b>	<b>Implementazione</b>	<b>22</b>
3.1	Struttura dei package . . . . .	22
3.2	Classi . . . . .	22
3.2.1	Classi in businessLogic . . . . .	22
3.2.2	Classi in dao . . . . .	25
3.2.3	Classi in domainModel . . . . .	29
<b>4</b>	<b>Testing</b>	<b>33</b>
4.1	White box testing . . . . .	33
4.1.1	LoginControllerTest . . . . .	33
4.1.2	CostumerControllerTest . . . . .	34
4.1.3	GymManagerControllerTest . . . . .	35
4.1.4	PersonalTrainerControllerTest . . . . .	36
4.1.5	ReceptionistControllerTest . . . . .	37
4.2	Black box testing . . . . .	37

# 1 Introduzione

## 1.1 Motivazione

Ho deciso di implementare un software per la gestione di una palestra perché da molto tempo frequento una struttura che nel corso di questi anni si è affidata ad un software in continuo aggiornamento. Il software per la gestione sia delle schede da parte dei personal trainer, sia dei clienti per la visualizzazione delle suddette è infatti cambiato negli anni. Gli anni trascorsi nella struttura, anche a stretto contatto con i personal trainer, mi hanno permesso di capire le funzionalità generali di un software di questo tipo e incuriosito dal capire quali fossero le problematiche riguardo all'implementazione di un software ho scelto di eseguire questo elaborato.

## 1.2 Statement

In questa applicazione da un lato i clienti possono visualizzare le loro schede di allenamento e i loro progressi, dall'altro il gestore della struttura e i vari dipendenti possono interagire attraverso un interfaccia che permette la corretta esecuzione delle varie operazioni. L'applicazione ha l'intenzione di essere sfruttata da più persone appartenenti a categorie diverse dove ognuna ha le proprie funzionalità:

- **Gestore della palestra.** Colui che ha la possibilità di aggiungere nuovi dipendenti al sistema (personal trainer e receptionist). Il gestore della palestra ha inoltre la possibilità di visualizzare i flussi di cassa per valutare i guadagni della struttura. Può visualizzare la lista degli abbonamenti e anche la lista di tutti i clienti. Il gestore della palestra è quindi colui che ha un accesso completo ai dati salvati all'interno del software.
- **Receptionist.** In questo ruolo si ha vari compiti, infatti un receptionist ha il dovere di registrare i clienti non appena si presentano alla struttura. Il receptionist deve illustrare i vari abbonamenti disponibili e registrare quindi l'abbonamento scelto dal cliente nel software. Se si tratta di un abbonamento a pagamento l'inserimento può essere effettuato solo se collegato ad uno scontrino fiscale, quindi il receptionist deve ricevere il pagamento e fare uno scontrino. Una volta che il cliente ha sottoscritto un abbonamento il receptionist consegna un badge che verrà utilizzato per l'accesso alla struttura. Il receptionist dialogando con il sistema assegna al badge l'identità del cliente in modo tale che il badge sia utilizzabile. Il cliente per accedere alla struttura avvicinerà il badge al sensore, al che il sistema deve fare visualizzare un messaggio al receptionist di accesso consentito o meno. Quindi il receptionist è responsabile all'ingresso dei clienti alla struttura.
- **Personal trainer.** Un personal trainer è colui che segue negli allenamenti i vari clienti. Il suo principale lavoro è quello di creare delle schede di allenamento per i clienti. Il personal trainer può inoltre fare delle misurazioni e delle valutazioni del cliente non appena lo ritiene opportuno, ad intervalli di tempo regolari oppure quando il cliente termina il ciclo di una scheda di allenamento. Il personal trainer affida ad ogni nuovo cliente una scheda predefinita fra quelle che dispone e che ha

creato in precedenza. La scelta fra quale scheda affidare al cliente può avvenire anche in seguito ad una prima valutazione del cliente con un "livello di forma". Il livello di forma è un numero che ha il solo scopo di fare capire indicativamente al personal trainer il livello del cliente per poter affidare più semplicemente una scheda di allenamento. Questo valore numerico fa parte di una valutazione che può contenere anche commenti. Alla valutazione è necessaria l'associazione di una serie di misurazioni quali il peso, la massa grassa, la massa magra etc... calcolati attraverso l'utilizzo di un Bioimpedenziometro. Oltre alle schede di allenamento di default il personal trainer quando lo ritiene opportuno può consegnare al cliente una scheda di allenamento personalizzata. Ogni scheda di allenamento ha la propria data di emissione e di scadenza, nonché il numero di livello di difficoltà che rispecchia il livello di forma eventualmente misurato con una valutazione. Il personal trainer deve poter visualizzare le schede di allenamento dei propri clienti, questo anche nell'ottica di poter copiare una vecchia scheda già creata per un cliente, modificarla in alcuni esercizi e renderla adatta ad un nuovo cliente.

- **Cliente.** Il cliente può visualizzare il proprio diario di allenamento dall'applicazione, quindi può visualizzare tutta la cronologia delle schede a lui proposte, e le varie valutazioni. Le valutazioni potranno anche essere visualizzate sotto forma di "grafico dei progressi" tramite il quale il cliente potrà osservare il proprio percorso. Oltre alle schede di allenamento il cliente potrà visualizzare la cronologia dei propri abbonamenti per osservare la data di scadenza e assicurarsi di poter sempre accedere alla struttura.

### 1.3 Estensioni future

All'applicazione potrebbero essere associate ulteriori funzionalità come:

- Gestione di una rete di palestre, in questo caso dovrebbe essere necessario associare ad ogni palestra i propri dipendenti e i propri clienti, il software centralizzato potrebbe permettere ingressi con lo stesso abbonamento ad una o all'altra palestra;
- Aggiunta della funzionalità "Corsi", se una palestra disponesse anche di corsi specifici oltre al solo servizio di sala pesi potrebbe essere implementata la sezione per la prenotazione ai corsi;
- Possibilità da parte del gestore di aggiungere nuove offerte di abbonamento attraverso un'interfaccia semplice;
- Gestione di un tornello per l'accesso alla struttura completamente automatizzato;

## 2 Analisi e progettazione

Il progetto è scritto nel linguaggio Java.

In questa versione del progetto il comportamento del database è imitato da una implementazione non persistente.

In questa versione del progetto l'interfaccia utente non è direttamente visionabile, ma nonostante questo il software espone già le funzioni necessarie per il corretto funzionamento dell'applicazione. Inoltre alcune viste potrebbero essere implementate facilmente seguendo i mock-ups allegati, utili per capire quali delle informazioni è possibile ricevere dal software.

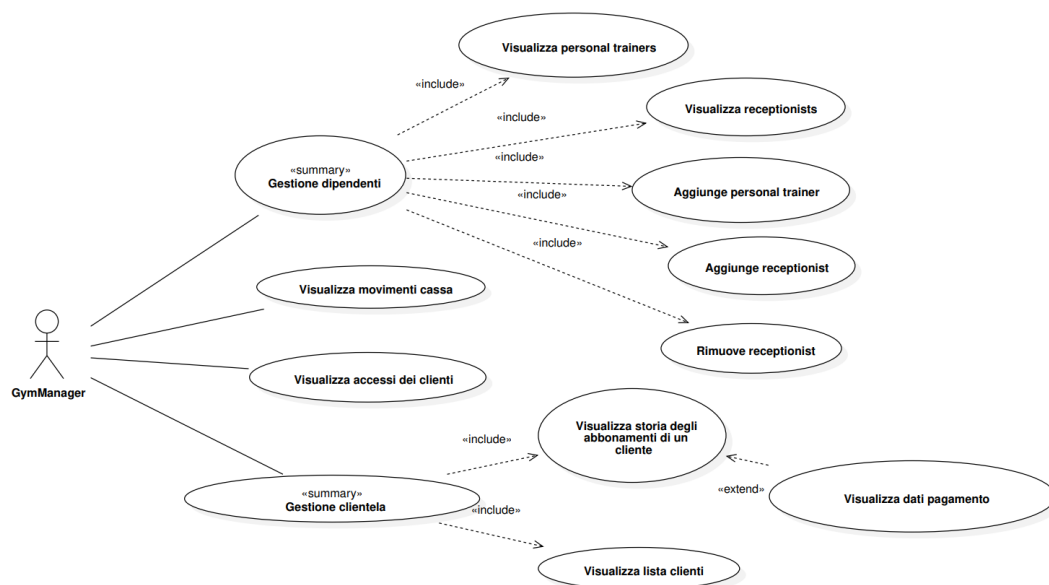
Il testing del progetto é effettuato con l'utilizzo di JUnit.

Link per il repo su GitHub: [repo](#). Da qui è possibile visionare tutto il codice.

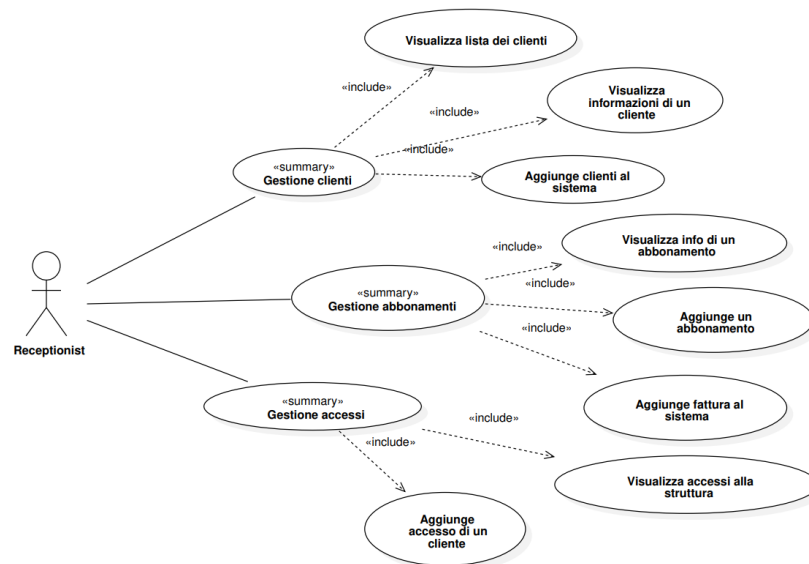
### 2.1 Use Case Diagram

Gli attori individuati sono i seguenti (per evitare una ripetizione di quanto detto in precedenza il commento ad ogni attore è minimale):

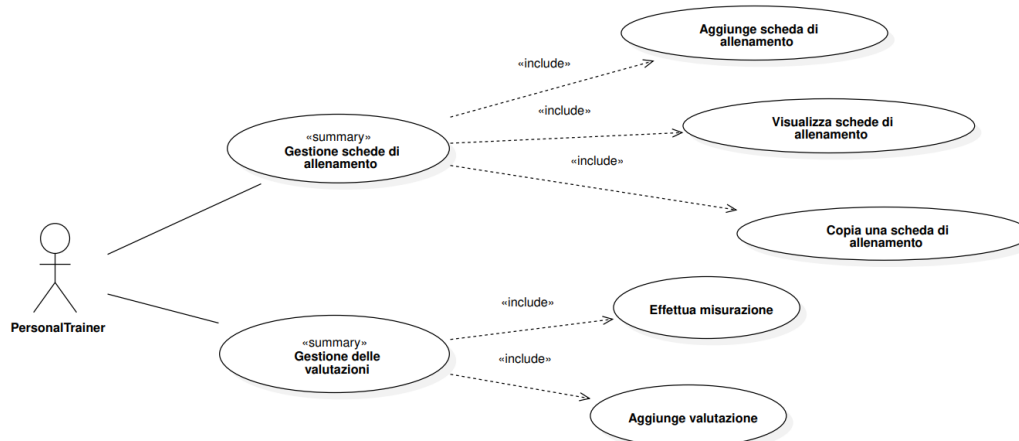
- **Gestore della palestra.** Il gestore della palestra è colui che può visionare tutte le informazioni della palestra, dalle informazioni dei clienti a quelle dei receptionist e dei personal trainer. Dal momento che il gestore della palestra è responsabile delle assunzioni di personal trainer e receptionist solo esso può aggiungere tramite la propria interfaccia nuovi dipendenti al sistema. Solo con l'aggiunta da parte del gestore della palestra infatti i dipendenti potranno entrare nel sistema ed interagire con esso.



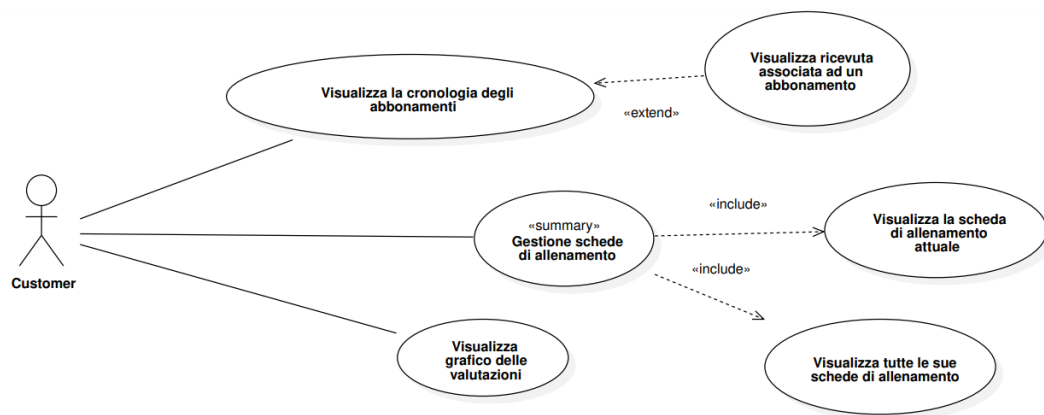
- **Receptionist.** I receptionist sono coloro che aggiungono il cliente al sistema, possono aggiungere manualmente un accesso alla palestra, oppure tramite il badge e possono visualizzare tutti i clienti e le proprie informazioni. I receptionist inoltre sono responsabili dell'aggiunta al sistema dell'abbonamento acquistato dal cliente, nonché della relativa "fattura".



- **Personal trainer.** I personal trainer sono responsabili della creazione e dell'aggiunta delle schede di allenamento, nonché delle valutazioni che redigono una volta completata una scheda di allenamento o quando essi lo ritengono necessario.



- **Customer.** Il cliente può visualizzare i suoi abbonamenti con i rispettivi dati. Può inoltre visualizzare le proprie schede di allenamento, vecchie ed attuale, nonché il grafico delle misurazioni e le varie valutazioni avute nel tempo.



## 2.2 Use Case Templates

Di seguito alcuni use case templates che descrivono gli user goals più importanti.

Use case	Description
Name	Visualizza una scheda di allenamento
Description	Visualizza gli esercizi e le informazioni su data di emissione, scadenza e del personal trainer di una scheda di allenamento scelta fra quelle disponibili
Actors	Customer
Level	User goal
Preconditions	Il customer deve avere un proprio profilo nel sistema
Normal flow	<ol style="list-style-type: none"><li>1 Il cliente accede al sistema. (mock up: 1)</li><li>2 Il sistema mostra una scelta fra la visualizzazione delle schede di allenamento, delle valutazioni o della cronologia degli abbonamenti del cliente. (mock up: 2)</li><li>3 Il cliente clicca sulla visualizzazione delle schede di allenamento.</li><li>4a Il sistema mostra tutte le schede di allenamento del cliente, con l'informazione sulla scadenza di ognuna. (mock up: 3)</li><li>5a Il cliente clicca su una scheda che desidera visualizzare.</li><li>6 Il sistema mostra le informazioni della scheda di allenamento, cioè gli esercizi, il personal trainer che ha creato la scheda, la data di emissione e di scadenza. (mock up: 4)</li></ol>
Alternative flows	<ol style="list-style-type: none"><li>4b Il cliente non ha al momento nessuna scheda da visualizzare.</li><li>5b Il sistema mostra il messaggio "Nessuna scheda da visualizzare". (mock up: 9)</li></ol>
Postconditions	Il cliente visualizza la scheda di allenamento o viene mostrato un messaggio di errore



Use case	Description
Name	Aggiungere un abbonamento al sistema
Description	Aggiunge un abbonamento fra le tipologie previste al sistema
Actors	Receptionist
Level	User goal
Preconditions	Il receptionist ha già effettuato il login ed esiste nel sistema.
Normal flow	<ol style="list-style-type: none"> <li>1 Il sistema mostra una scelta fra aggiungere abbonamento, aggiungere accesso e visualizzare la lista dei clienti. (mock up: 5)</li> <li>2 Il receptionist clicca su "Aggiungi abbonamento".</li> <li>3 Il sistema mostra la schermata di aggiunta di un abbonamento. (mock up: 5)</li> <li>4 Il receptionist cerca il customer nel sistema.</li> <li>5a Il receptionist seleziona il customer cercato.</li> <li>6a Il receptionist seleziona la data di emissione e il tipo di abbonamento (accesso singolo o no).</li> <li>7 Il receptionist riceve il pagamento.</li> <li>8 Il receptionist aggiunge il pagamento al sistema.</li> <li>9 Il sistema automaticamente inserisce il numero di ricevuta nell'apposito spazio.</li> <li>10 Il receptionist clicca su "Confirm".</li> <li>11 Il sistema mostra un messaggio di aggiunta completata.</li> </ol>
Alternative flows	<ol style="list-style-type: none"> <li>5b Il cliente cercato non esiste nel sistema.</li> <li>6b Il sistema mostra il messaggio "Nessun cliente da visualizzare, assicurarsi di avere inserito il cliente nel sistema". (mock up: 9)</li> </ol>
Postconditions	L'abbonamento o l'ingresso singolo sono aggiunti al sistema e il cliente può fare accesso alla palestra o viene mostrato un messaggio di errore.

Use case	Description
Name	Aggiunta di un accesso alla palestra
Description	L'accesso di un cliente alla palestra viene registrato
Actors	Receptionist
Level	User goal
Preconditions	
Normal flow	<p>1a Il cliente passa il badge nel sensore predisposto.</p> <p>2a Il sensore riconosce l'id del badge.</p> <p>3a Il sistema cerca nel database il cliente.</p> <p>4a Il sistema trova il cliente nel database.</p> <p>5a Il sistema cerca gli abbonamenti del cliente.</p> <p>6a Il sistema trova un abbonamento valido.</p> <p>7a Il sistema espone un messaggio di accesso consentito. (stile del mock-up: 9)</p>
Alternative flows	<p>1b Il cliente non ha con se il badge.</p> <p>2b Il sistema mostra al receptionist la schermata principale. (mock up: 5)</p> <p>3b Il receptionist clicca su "Aggiungi accesso".</p> <p>4b Il sistema mostra la schermata di aggiunta dell'accesso. (mock up: 6)</p> <p>5b Il receptionist cerca il cliente nel sistema.</p> <p>6b Il receptionist seleziona il cliente.</p> <p>7b Il receptionist clicca su "Confirm".</p> <p>6c Il cliente non è nel sistema.</p> <p>7c Il sistema mostra il messaggio "Nessun cliente da visualizzare". (mock up: 9)</p> <p>4d Il sistema non trova l'id collegato al cliente e mostra il messaggio "Badge non riconosciuto". (mock-up: 9)</p>
Postconditions	Viene inserito un accesso o viene mostrato un messaggio di errore.

Use case	Description
Name	Aggiunta di una scheda di allenamento personalizzata al cliente
Description	Aggiunta di una scheda di allenamento ed associazione al cliente in modo tale che questo possa visualizzarla
Actors	Personal trainer
Level	User goal
Preconditions	Il personal trainer ha effettuato l'accesso al sistema. Il cliente è già stato inserito nel sistema.
Normal flow	<ol style="list-style-type: none"> <li>1 Il sistema mostra la schermata principale del personal trainer. (mock up: 7)</li> <li>3 Il personal trainer clicca su "Aggiungi scheda di allenamento".</li> <li>4 Il sistema mostra una schermata di aggiunta di una scheda di allenamento. (mock up: 8)</li> <li>5 Il personal trainer immette gli esercizi della scheda di allenamento.</li> <li>6 Il personal trainer inserisce le informazioni dell'emissione della scheda e della sua scadenza.</li> <li>7 Il personal trainer ricerca il cliente nel sistema.</li> <li>8 Il personal trainer seleziona il cliente.</li> <li>9 Il personal trainer clicca su "Confirm".</li> <li>10 Il sistema mostra un messaggio di avvenuta aggiunta della scheda. (mock up: 9)</li> </ol>
Alternative flows	<ol style="list-style-type: none"> <li>5b Il personal trainer clicca sul comando "Copia da scheda già creata". (mock up: 8)</li> <li>6b Il sistema mostra le schede già create dal personal trainer.</li> <li>7b Il personal trainer seleziona una scheda già creata.</li> <li>8b Il sistema compila i campi con i dati della scheda copiata ad eccezione del customer.</li> <li>9b Il personal trainer modifica il customer.</li> <li>10b Il personal trainer può modificare gli esercizi.</li> </ol>
Postconditions	Viene assegnata una scheda di allenamento al cliente.

## 2.3 Mock ups

Di seguito una serie di mock ups che rappresentano schematicamente come alcune delle finestre più importanti nell'applicazione possano essere rappresentate. Sorvoliamo in questo caso dal design dell'applicazione che può essere sviluppata in un secondo momento, ma questi sono importanti per capire meglio quali sono le funzionalità e le informazioni che è possibile avere all'interno del sistema.

### 2.3.1 LoginPage

The mockup shows a window titled "Main Window" with a login form. It includes a label "I am..." next to a dropdown menu currently showing "Costumer" with a downward arrow. The dropdown menu is open, showing three options: "Gym Manager", "Personal Trainer", and "Receptionist". Below the dropdown are three text input fields labeled "Name", "Surname", and "PhoneNumber".

Figure 1: Schermata per il "login"

### 2.3.2 Cliente

The mockup shows a window titled "Costumer: Tommaso Botarelli". Inside the window, there are three buttons stacked vertically, labeled "My training card", "My progress", and "My subscription".

Figure 2: Finestra principale del cliente

Costumer: Tommaso Botarelli






Training card's name	Emission	Expiration	Personal trainer's name
 <a href="#">Click for more information</a>			


Figure 3: Finestra per la visualizzazione delle proprie schede di allenamento


Costumer: Tommaso Botarelli

### Training card's name

Personal trainer's name:

Emission:  

Expiration:  

Level:  

Exercise 1

Exercise 2




Figure 4: Finestra della visualizzazione di una scheda di allenamento

### 2.3.3 Receptionist

Receptionist: receptionist's name

Costumer:

Type of access:

Type of sub:

Emission:

BillID:

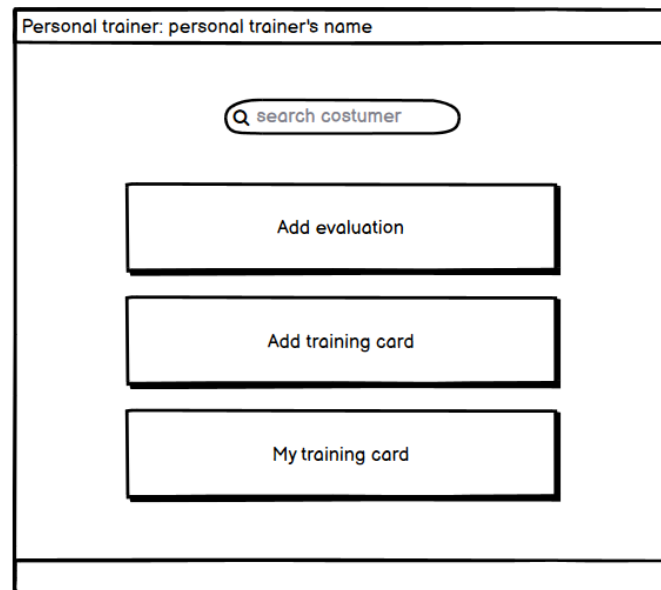
Figure 5: Finestra principale del receptionist

Receptionist: name of receptionist

Costumer:

Figure 6: Finestra per l'aggiunta di un accesso

### 2.3.4 Personal trainer



Personal trainer: personal trainer's name

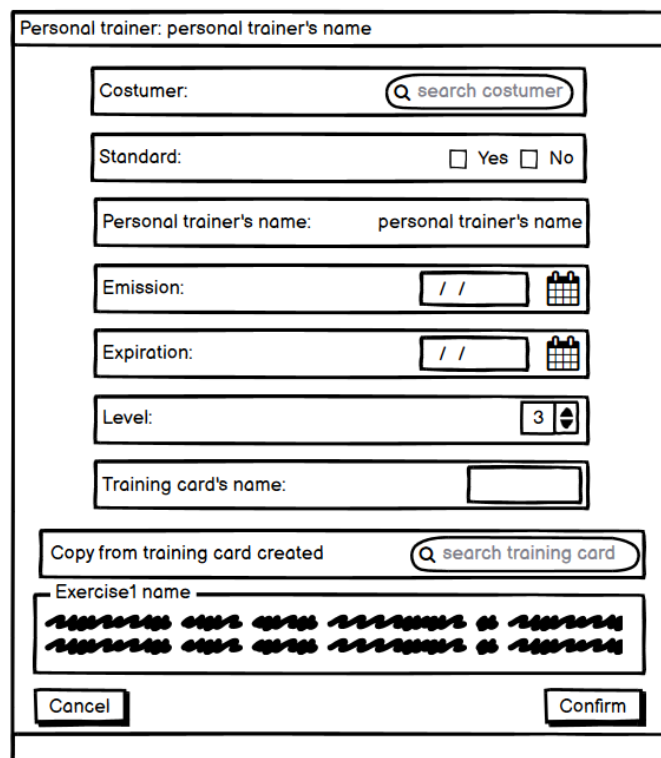
Q search costumer

Add evaluation

Add training card

My training card

Figure 7: Finestra principale personal trainer





Personal trainer: personal trainer's name


Costumer: Q search costumer

Standard: ☐ Yes ☐ No

Personal trainer's name: personal trainer's name

Emission: / / 

Expiration: / / 

Level: 3 

Training card's name:

Copy from training card created Q search training card

Exercise1 name

Cancel Confirm

Figure 8: Finestra per l'aggiunta di una scheda di allenamento

### 2.3.5 Varie

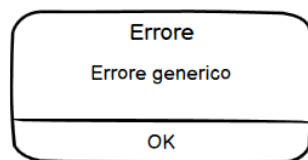
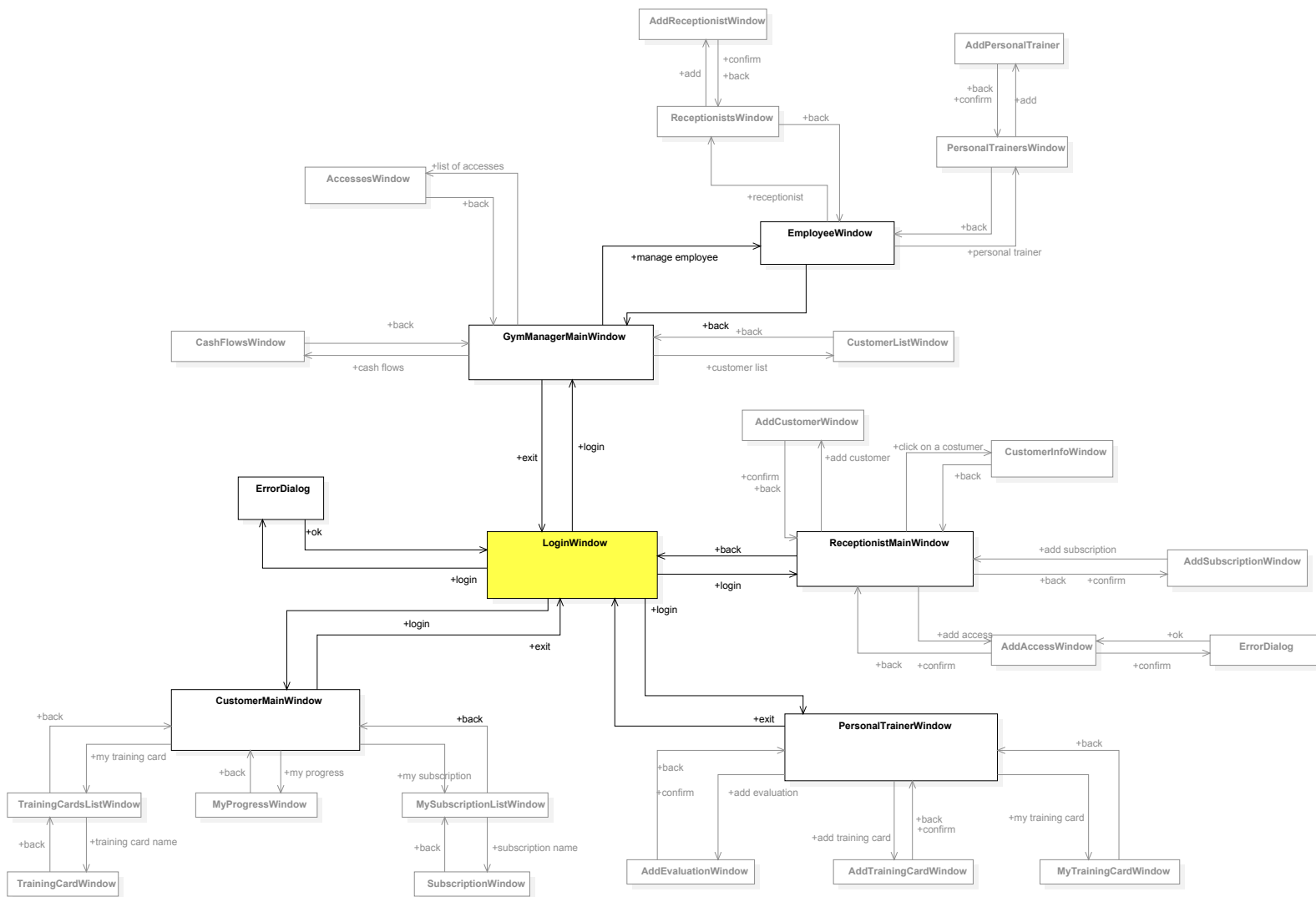


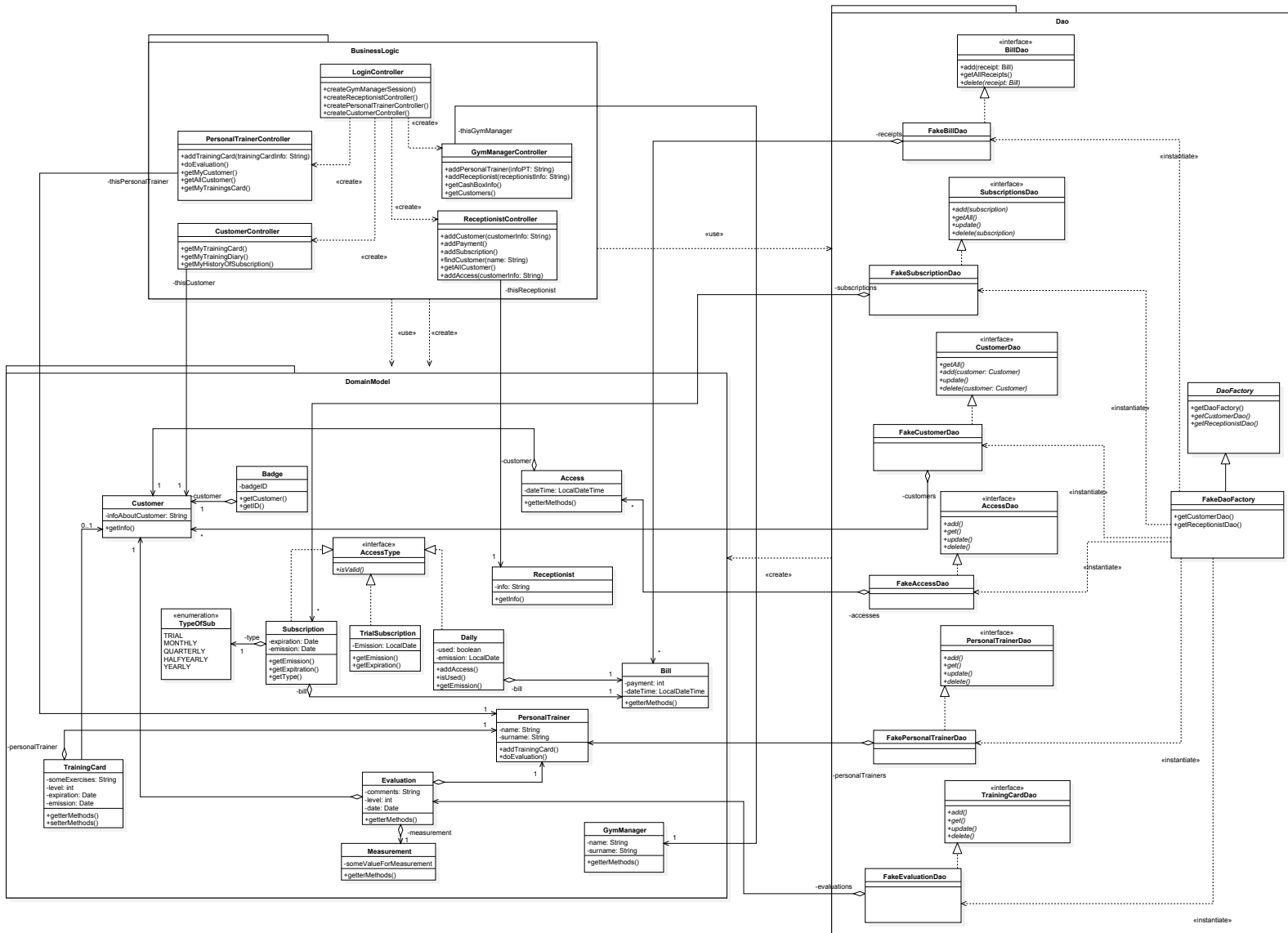
Figure 9: Finestra di errore/avviso generico



## 2.4 Page navigation diagram



## 2.5 Class diagram



## 2.6 Pattern utilizzati

All'interno del progetto vengono usati i seguenti patter:

- Abstract factory
- Singleton

### 2.6.1 Abstract factory

Il pattern abstract factory viene utilizzato nella forma illustrata nel GOF:

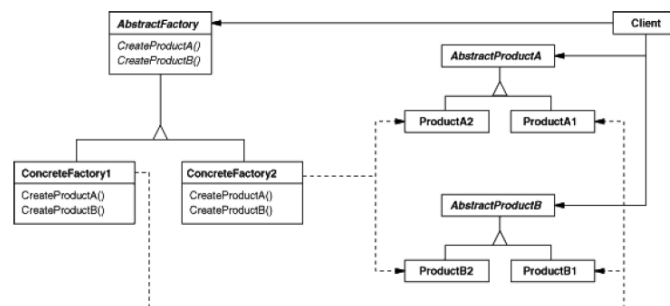


Figure 10: Struttura del pattern abstract factory nel GOF

Dove nel caso di questo progetto, assume la seguente struttura:

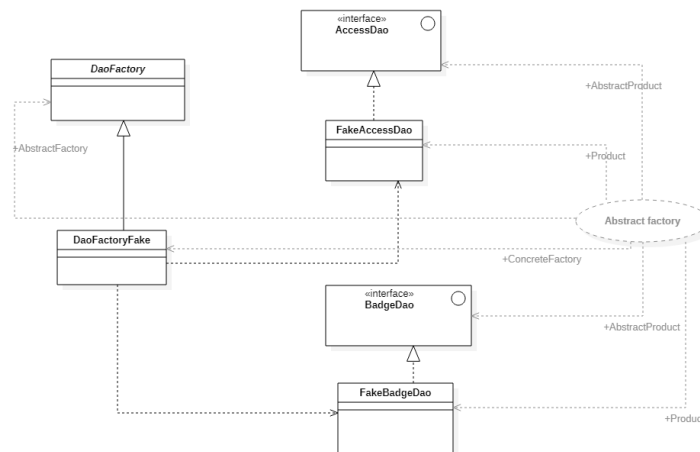


Figure 11: Struttura del pattern abstract factory in questo progetto

Nella figura sopra per una visualizzazione migliore sono state rimosse molte delle interfacce e le relative implementazioni relativi a tutti i dao presenti.

Come già accennato in precedenza l'utilizzo del pattern abstract factory è volto alla semplicità con la quale una implementazione vera di un database può essere aggiunta. Volendo infatti aggiungere un database effettivo all'applicazione la struttura potrebbe diventare la seguente:

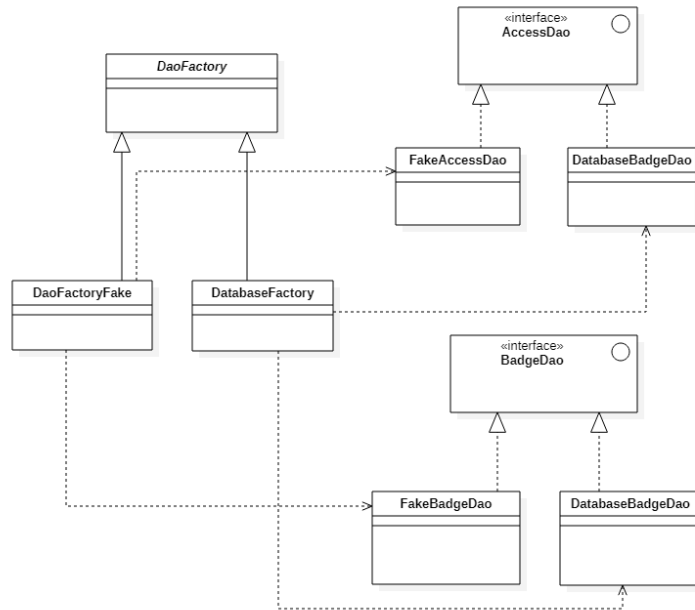


Figure 12: Struttura del pattern abstract factory con l'aggiunta del database

A questo punto per rendere effettiva la modifica basterebbe modificare l'unico metodo non astratto della classe *DaoFactory* introducendo il caso di selezione del database:

```

public static DaoFactory getDaoFactory(int nFactory){
    switch (nFactory){
        case FAKE:
        {
            if (instance == null)
                instance = new DaoFactoryFake();
            return instance;
        }
        default:
            return null;
    }
}

```

Figure 13: Attuale metodo *getDaoFactory*

L'utilizzo di questo pattern è quindi giustificato dalla semplicità con la quale il software può essere esteso con nuove funzionalità, proprio come indicato dal principio "open closed", cioè aperto all'estensione e chiuso alla modifica.

## 2.6.2 Singleton

Il pattern singleton è implementato nel seguente modo:

```

public class FakeAccessDao implements AccessDao {

    ArrayList<Access> acces;
    private static FakeAccessDao instance = null;

    private FakeAccessDao() { acces = new ArrayList<>(); }

    public static FakeAccessDao getInstance(){
        if (instance == null){
            instance = new FakeAccessDao();
        }
        return instance;
    }
}

```

Figure 14: Esempio di singleton usato nel progetto

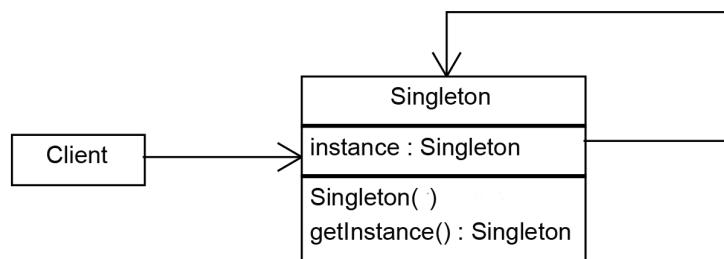


Figure 15: Struttura del pattern Singleton

Questo pattern è usato da tutte le factory concrete che si trovano nel sistema. Questo per fare in modo che si abbia coerenza con i dati inseriti nei relativi Dao. Nel caso infatti non si usasse questo pattern ogni nuova istanza di un dao avrebbe una lista di dati vuota.

Facendo in questo modo invece ogni volta che si richiede una istanza di un FakeDao si ha come riferimento un FakeDao già creato e usato da tutto il sistema, i dati quindi sono mantenuti aggiornati fra i vari componenti del software.

## 3 Implementazione

### 3.1 Struttura dei package

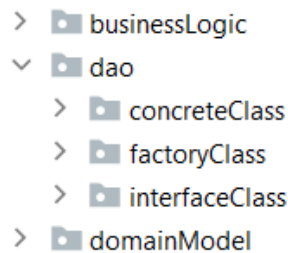


Figure 16: Struttura dei packages

Come si può vedere in figura sopra i packages individuati sono i seguenti:

- businessLogic, package che contiene i controller che il presentation layer può utilizzare per mostrare le informazioni a schermo;
- dao, package che è diviso a sua volta in tre packages:
  - factoryClass, contiene le classi utilizzate per implementare il pattern abstract factory;
  - interfaceClass, contiene le classi che fanno da interfaccia alle concrete class del pattern factory;
  - concreteClass, contiene le classi concrete del pattern abstract factory;
- domainModel, contiene tutte le classi che fanno parte del dominio;

### 3.2 Classi

In questa sezione vengono riportate le classi di maggior importanza nel sistema con una piccola spiegazione del loro utilizzo e dei metodi più importanti.

#### 3.2.1 Classi in businessLogic

All'interno di questo package le classi che troviamo sono le seguenti:

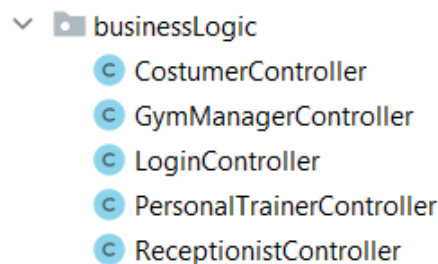


Figure 17: Classi all'interno del package "businessLogic"

**LoginController** Questa classe è costruita per esporre i metodi necessari per creare una sessione utente all'interno del software. Le sessioni che si possono creare sono relative al cliente, al gestore della palestra, al receptionist e al personal trainer.

Possiamo soffermarci sul metodo per la creazione della sessione di un cliente (gli altri metodi hanno funzionamento presso a poco analogo):

```
public CustomerController createCustomerSession(String name, String surname, String phoneNumber) throws Exception{
    Customer user = customerDao.getSelectedCustomer(name, surname, phoneNumber);
    if (user != null) {
        CustomerController controller = new CustomerController();
        controller.setCurrentUser(user);
        return controller;
    }
    else
        throw new Exception("A customer with this data doesn't exist");
}
```

Il metodo utilizza il dao di riferimento per avere indietro il cliente associato a quei dati. Se nessun customer all'interno del sistema ha quei dati (nome, cognome e numero di telefono) allora viene lanciata una eccezione.

Se da una parte questo metodo di "login" è sicuramente inefficiente, (basterebbe conoscere nome, cognome e numero di telefono di un individuo per accedere al suo "account") dall'altra è utile per capire almeno a grandi linee il funzionamento di una schermata di login.

Se pensiamo al funzionamento in una applicazione completa di presentation layer possiamo immaginare come quest'ultimo possa utilizzare questa classe nel seguente modo:

- Il metodo restituisce un *customerController*, allora si può passare alla vista del menù principale del cliente;
- Il metodo lancia una eccezione, viene mostrato un errore che informa sul fatto che non esista un cliente con i dati inseriti;

Questa classe avrà quindi il riferimento ai dao dei clienti, dei gestori della palestra, dei receptionist e dei personal trainer per permettere il "login" oppure no.

```
public class LoginController {
    private CustomerDao customerDao;
    private ReceptionistDao receptionistDao;
    private PersonalTrainerDao personalTrainerDao;
    private GymManagerDao gymManagerDao;

    public LoginController(){...}

    public CustomerController createCustomerSession(String name, String surname, String phoneNumber) throws Exception{...}

    public ReceptionistController createReceptionistSession(String name, String surname, String phoneNumber) throws Exception{...}

    public PersonalTrainerController createPersonalTrainerSession(String name, String surname, String phoneNumber) throws Exception{...}

    public GymManagerController createGymManagerSession(String name, String surname, String phoneNumber) throws Exception{...}
}
```

**GymManagerController** Prendiamo come esempio dei rimanenti controller quello relativo al gestore della palestra. Questa classe come le rimanenti è utilizzata per fornire tutte le funzioni necessarie a completare i casi d'uso del rispettivo user. Nel caso di *GymManagerController* quindi ci saranno metodi per l'aggiunta di receptionist e personal trainer, per vedere gli scontrini del giorno e tutti gli scontrini e così via...

```
public ArrayList<Bill> getAllBills() { return billDao.getAll(); }

public ArrayList<Bill> getBillsOfTheDay(LocalDate date) { return billDao.getFromDate(date); }

public ArrayList<Access> getAllAccess() { return accessDao.getAll(); }

public ArrayList<Access> getAllAccessFromDate(int year, int month, int day){
    return accessDao.getFromDate(LocalDate.of(year, month, day));
}

public ArrayList<Costumer> getAllCostumers() { return costumerDao.getAll(); }

public ArrayList<Costumer> searchCostumerFromNameSurname(String name, String surname){...}

public ArrayList<AccessType> getSubOfCostumer(Costumer costumer){...}

public ArrayList<Receptionist> getAllReceptionist() { return receptionistDao.getAllReceptionist(); }

public void addReceptionist(String name, String surname, String phoneNumber){...}
```

Figure 18: Alcuni metodi di *GymManagerController*

Questa classe come si può vedere presenta molti metodi che sono semplici chiamate al dao relativo al dato che si vuole presentare. Questo perché il manager della palestra è colui che ha il controllo su tutti i dati e la possibilità di avere tutte le informazioni della palestra.

```
public ArrayList<Bill> getAllBills(){
    return billDao.getAll();
}

public ArrayList<Bill> getBillsOfTheDay(LocalDate date){
    return billDao.getFromDate(date);
}

public ArrayList<Access> getAllAccess(){
    return accessDao.getAll();
}

public ArrayList<Access> getAllAccessFromDate(int year, int month, int day){
    return accessDao.getFromDate(LocalDate.of(year, month, day));
}
```

Figure 19: Dettaglio di alcuni metodi di *GymManagerController*



### 3.2.2 Classi in dao

Le classi all'interno di questo package sono utilizzate dalle classi della businessLogic per interfacciarsi con il database. Ogni classe concreta dei dao serve quindi teoricamente ad interfacciarsi con il database (aggiunta di nuove informazioni, richiesta di dati), ma in questa versione il database è semplicemente imitato da classi concrete che mantengono i dati in *ArrayList*.

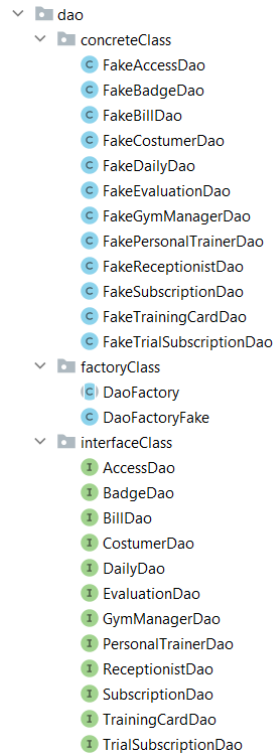


Figure 20: Tutte le classi nel package dao

**factoryClass** Questo package contiene due classi:

- **DaoFactory**, la classe astratta utilizzata per l'implementazione del pattern abstract factory. Questa classe ha la funzione (come spiegato meglio successivamente) di rendere una istanza della factory del tipo specificato in *getDaoFactory(int nFactory)*. Per il momento l'unico tipo che si può richiedere è quello *FAKE*, tuttavia pensando ad una eventuale estensione dell'applicazione e l'introduzione di un database vero, con questa struttura delle classi appare molto facile l'introduzione di nuove classi.

```

public abstract class DaoFactory {
    public static final int FAKE = 1;

    private static DaoFactoryFake instance = null;

    public abstract CostumerDao getCostumerDao();
    public abstract AccessDao getAccessDao();
    public abstract BadgeDao getBadgeDao();
    public abstract BillDao getBillDao();
    public abstract EvaluationDao getEvaluationDao();
    public abstract PersonalTrainerDao getPersonalTrainerDao();
    public abstract ReceptionistDao getReceptionistDao();
    public abstract TrainingCardDao getTrainingCardDao();
    public abstract GymManagerDao getGymManagerDao();
    public abstract DailyDao getDailyDao();
    public abstract SubscriptionDao getSubscriptionDao();
    public abstract TrialSubscriptionDao getTrialSubscriptionDao();

    public static DaoFactory getDaoFactory(int nFactory){...}
}

public static DaoFactory getDaoFactory(int nFactory){
    switch (nFactory){
        case FAKE:
        {
            if (instance == null)
                instance = new DaoFactoryFake();
            return instance;
        }
        default:
            return null;
    }
}

```

Figure 21: Il metodo *getDaoFactory*

Come si può vedere dall'immagine sopra il metodo *getDaoFactory* è il metodo che permette la selezione della factory da usare. Questo è il metodo alla quale aggiungere nuovi "case" per implementare un database reale (più info nella sezione "Pattern").

- **DaoFactoryFake**, come verrà spiegato nella sezione "Pattern" in seguito, questa è la classe utilizzata per avere un istanza dei dao "fake", cioè quei dao previsti da questa versione del sistema software per la quale i dati sono salvati in modo non permanente in strutture dati come gli *ArrayList*. Questa classe fornisce quindi le classi concrete (che mantengono i metodi dati dalle relative interfacce) dei dao di tipo *FAKE*.

```

public class DaoFactoryFake extends DaoFactory{

    @Override
    public CostumerDao getCostumerDao() { return FakeCostumerDao.getInstance(); }

    @Override
    public AccessDao getAccessDao() { return FakeAccessDao.getInstance(); }

    @Override
    public BadgeDao getBadgeDao() { return FakeBadgeDao.getInstance(); }
}

```

Figure 22: Alcuni metodi della classe

**interfaceClass** Questo package mantiene le interfacce che si possono vedere dalla figura 3.2.2. Le interfacce in questo package sono importanti per lo sviluppo del pattern dao, tutte le classi concrete (del tipo "fake", ma anche di altri tipi che possono essere aggiunti in un secondo momento) implementeranno la relativa interfaccia per avere gli stessi metodi esposti.

Prendendo come esempio di questo gruppo l'interfaccia relativa al dao del cliente (**CostumerDao**) si può vedere come questa esponga un metodo per l'aggiunta e la rimozione di un cliente e per ricevere tutti i clienti (sia senza condizioni che con un filtro basato sul nome e il cognome).

```

public interface CostumerDao {
    void add(Costumer costumer);
    boolean delete(Costumer costumer);
    ArrayList<Costumer> getAll();
    ArrayList<Costumer> getFromNameSurname(String name, String surname);
    Costumer getSelectedCostumer(String name, String surname, String mobilePhone);
    void deleteAll();
}

```

**concreteClass** In questo package troviamo le classi concrete che implementano le interfacce viste sopra. Per il momento come già detto all'interno di questo package troviamo solo classi concrete del tipo "fake", quindi ognuna di queste classi non ha un riferimento al database, ma in realtà ha una memorizzazione non permanente dei dati in strutture dati mantenute nella classe stessa. Quindi prendendo come esempio la classe relativa al dao del customer (**FakeCostumerDao**) essa presenterà l'implementazione dei metodi visti prima. Da notare come questa classe e le altre classi in questo package abbiano il costruttore privato. Questo consente di ottenere un'istanza della classe tramite il metodo statico pubblico *getInstance()* metodo che prima controlla se una istanza della classe è già stata costruita, in questo caso verrà reso un riferimento a tale istanza, in caso contrario verrà costruita una nuova istanza (pattern **singleton**).

Vediamo quindi la classe ed alcuni metodi:

```

public class FakeCostumerDao implements CostumerDao {

    private ArrayList<Costumer> costumers;
    private static FakeCostumerDao instance = null;

    public static FakeCostumerDao getInstance(){
        if (instance == null){
            instance = new FakeCostumerDao();
        }
        return instance;
    }

    private FakeCostumerDao() { costumers = new ArrayList<>(); }

    @Override
    public void add(Costumer costumer) {...}

    @Override
    public boolean delete(Costumer costumer) {...}

    @Override
    public ArrayList<Costumer> getAll() { return costumers; }

    @Override
    public ArrayList<Costumer> getFromNameSurname(String name, String surname) {
        ArrayList<Costumer> returnArray = new ArrayList<>();

        for (Costumer costumer : costumers){
            if(costumer.getName().equals(name) && costumer.getSurname().equals(surname))
                returnArray.add(costumer);
        }

        return returnArray;
    }

    @Override
    public Costumer getSelectedCostumer(String name, String surname, String phoneNumber) {
        Costumer returnCostumer = null;
        for (Costumer c : costumers){
            if (c.getName().equals(name) && c.getSurname().equals(surname) && c.getPhoneNumber().equals(phoneNumber))
                returnCostumer = c;
        }
        return returnCostumer;
    }

    @Override
    public void deleteAll() { costumers.clear(); }
}

```

### 3.2.3 Classi in domainModel

All'interno di questo package troviamo tutte le classi che formano il modello:



In generale queste classi hanno in comune il fatto di non avere logica all'interno. Infatti, a parte alcuni casi che verranno discussi, esse sono utilizzate come contenitori di informazioni e quindi i metodi che si ritrovano sono in generale dei *getter* e dei *setter*.

Una classe generica per questo tipo di package è la seguente:

```
public class Costumer {

    private String name;
    private String surname;
    private String phoneNumber;

    public Costumer(String name, String surname, String phoneNumber){
        this.name = name;
        this.surname = surname;
        this.phoneNumber = phoneNumber;
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getSurname() { return surname; }

    public void setSurname(String surname) { this.surname = surname; }

    public String getPhoneNumber() { return phoneNumber; }

    public void setPhoneNumber(String phoneNumber) { this.phoneNumber = phoneNumber; }

    @Override
    public boolean equals(Object obj) {
        return this.name.equalsIgnoreCase(((Costumer)obj).getName()) &&
            this.surname.equalsIgnoreCase(((Costumer)obj).getSurname()) &&
            this.phoneNumber.equalsIgnoreCase(((Costumer)obj).getPhoneNumber());
    }
}
```

Quindi come si vede da questo esempio si hanno delle informazioni (in questo caso il nome, il cognome e il numero di telefono) e poi dei getter e dei setter per impostare questi valori. Il metodo *equals()* è necessario per un corretto confronto fra Costumer.

Fra le classi più interessanti all'interno di questo package ci sono quelle relative ai tipi di abbonamento con la quale si può fare accesso alla palestra:

- **AccessType**, interfaccia che espone un solo metodo. Questa classe fa da interfaccia a tutti i tipi di abbonamento che permettono l'accesso alla palestra.

```
public interface AccessType {  
    boolean isValid(LocalDate date);  
}
```

- **TrialSubscription**, classe che identifica gli abbonamenti di prova della palestra e quindi quelli che hanno durata di 14 giorni e un massimo di 3 accessi. Questa classe implementa l'interfaccia AccessType, di conseguenza l'implementazione del metodo *isValid()* ritornerà vero se e solo se sono stati effettuati meno di 3 accessi con quell'abbonamento e allo stesso momento la data con la quale si tenta l'accesso è compresa fra il rilascio e la scadenza dei 14 giorni (compresi).

```
public TrialSubscription(Costumer costumer, LocalDate date){  
    this.costumerTarget = costumer;  
    this.emission = date;  
    this.expiration = date.plusDays(TypeOfSub.TRIAL.getnDay());  
    this.nAccess = 0;  
}
```

Figure 23: Costruttore di TrialSubscription

```
@Override  
public boolean isValid(LocalDate date){  
    return (date.isEqual(this.emission) || date.isEqual(this.expiration) ||  
            date.isAfter(this.emission) && date.isBefore(this.expiration)) &&  
            this.nAccess < 3;  
}
```

Figure 24: Implementazione del metodo *isEqual()*

- **Daily**, questa classe rappresenta l'abbonamento di singolo ingresso alla palestra, quindi dopo il primo accesso con un daily esso non potrà più essere utilizzato.

```
public Daily(LocalDate emission, Costumer costumer, Bill bill){  
    this.emission = emission;  
    this.used = false;  
    this.myCostumer = costumer;  
    this.bill = bill;  
}
```

Figure 25: Costruttore di Daily

```
@Override
public boolean isValid(LocalDate date) { return date.isEqual(this.emission) && !this.used; }
```

Figure 26: Implementazione del metodo *isEqual()*

- **Subscription**, questa classe rappresenta l'abbonamento alla palestra che può essere di vari tipi: mensile, trimestrale, semestrale e annuale. Questa differenza è rappresentata dalla specifica del tipo selezionabile da una enumeration chiamata **TypeOfSub**.

```
TRIAL( nMonth: 0, nDay: 14, cost: 0),
MONTHLY( nMonth: 1, nDay: 0, cost: 50),
QUARTERLY( nMonth: 3, nDay: 0, cost: 120),
HALFYEARLY( nMonth: 6, nDay: 0, cost: 200),
YEARLY( nMonth: 12, nDay: 0, cost: 350);

final int nMonth;
final int nDay;
final float cost;

TypeOfSub(int nMonth, int nDay, float cost){
    this.nMonth = nMonth;
    this.nDay = nDay;
    this.cost = cost;
}

TypeOfSub(TypeOfSub typeOfSub){
    this.nDay = typeOfSub.nDay;
    this.nMonth = typeOfSub.nMonth;
    this.cost = typeOfSub.cost;
}
```

Figure 27: Enumerazione TypeOfSub

La classe Subscription utilizza questa enumerazione nel seguente modo:

```
public class Subscription implements AccessType{

    private LocalDate emission;
    private LocalDate expiration;
    private TypeOfSub type;
    private Costumer myCostumer;
    private Bill bill;

    public Subscription(LocalDate emission, TypeOfSub type, Costumer costumer, Bill bill){
        this.emission = emission;
        this.type = type;
        this.myCostumer = costumer;
        this.expiration = emission.plusMonths(type.getnMonth());
        this.bill = bill;
    }
}
```

Figure 28: Classe Subscription

Quindi utilizzando la validità di ogni singolo tipo di abbonamento si imposta la scadenza a partire dalla data di emissione. In questo caso il metodo *isValid()* controlla solo che la data con la quale si prova a fare accesso rientra nel range fra emissione e scadenza (compresi)

```
@Override
public boolean isValid(LocalDate date) {
    return date.isEqual(this.emission) || date.isEqual(this.expiration) ||
           date.isAfter(this.emission) && date.isBefore(this.expiration);
}
```

Figure 29: Implementazione del metodo *isValid()*



## 4 Testing

Il testing del progetto è stato eseguito in due prospettive distinte:

- White box
- Black box

### 4.1 White box testing

Questa sezione di testing si è concentrata nel testare le funzioni della business logic. Questo viene eseguito avendo una conoscenza della struttura del software e di come i dati vengono salvati.

Di ogni classe vengono di seguito analizzati i test più significativi/rappresentativi di essa. Per una completa comprensione di tutti i metodi testati si rimanda direttamente al codice.

#### 4.1.1 LoginControllerTest

La classe che testa questa suite è *LoginController*. Dal momento che i metodi di questa classe sono fra di loro analoghi (cambia il tipo di dato), prendendo come esempio il testing del metodo *createCostumerSession* si può avere una panoramica completa:

```
@Test
void createCostumerSession() {
    try {
        CostumerController goodCostumer = loginController.createCostumerSession( name: "Tommaso",
            surname: "Botarelli",
            phoneNumber: "56751862798"
        );
        assertNotNull(goodCostumer);
        CostumerController badCostumer = loginController.createCostumerSession( name: "Gianluca",
            surname: "Rossi",
            phoneNumber: "865736789"
        );
    }
    catch (Exception e){
        assertEquals( expected: "A costumer with this data doesn't exist", e.getMessage());
    }
}
```

Figure 30: Test del metodo *createCostumerSession()*

### 4.1.2 CostumerControllerTest

Viene testato in questo caso la classe CostumerController. Di questa classe sicuramente un metodo molto importante è quello che garantisce al cliente la visualizzazione delle proprie schede di allenamento. Il test eseguito è il seguente:

```
@Test
void getListOfMyTrainingCard() {
    Costumer costumer = new Costumer( name: "Tommaso", surname: "Botarelli", phoneNumber: "123456789");
    costumerDao.add(costumer);

    costumerController.setCurrentUser(costumer);

    PersonalTrainer personalTrainer = new PersonalTrainer( name: "Sandro", surname: "Giusti", phoneNumber: "763581610");
    personalTrainerDao.add(personalTrainer);

    TrainingCard trainingCard1 = new TrainingCard( exercises: "Some exercises", level: 2, standard: false, personalTrainer);
    TrainingCard trainingCard2 = new TrainingCard( exercises: "Some exercises", level: 5, standard: false, personalTrainer);
    trainingCard1.setCostumer(costumer);
    trainingCard2.setCostumer(costumer);

    trainingCardDao.addTrainingCard(trainingCard1);
    trainingCardDao.addTrainingCard(trainingCard2);

    ArrayList<TrainingCard> trainingCards = new ArrayList<>();

    trainingCards.add(trainingCard1);
    trainingCards.add(trainingCard2);

    assertEquals(trainingCards, costumerController.getListOfMyTrainingCard());
}
```

Figure 31: Test del metodo *getListOfMyTrainingCard()*

Per questa classe di test (ma analogamente anche per le seguenti) risultano fondamentali il setup prima dell'esecuzione dei test. Questo permette la creazione dei dao che la classe di testing utilizza per confermare i risultati con quello che ci aspettiamo.

```
@BeforeAll
static void setUp(){
    trialSubscriptionDao = Objects.requireNonNull(DaoFactory.getDaoFactory(1)).getTrialSubscriptionDao();
    subscriptionDao = Objects.requireNonNull(DaoFactory.getDaoFactory(1)).getSubscriptionDao();
    dailyDao = Objects.requireNonNull(DaoFactory.getDaoFactory(1)).getDailyDao();
    costumerDao = Objects.requireNonNull(DaoFactory.getDaoFactory(1)).getCostumerDao();
    trainingCardDao = Objects.requireNonNull(DaoFactory.getDaoFactory(1)).getTrainingCardDao();
    personalTrainerDao = Objects.requireNonNull(DaoFactory.getDaoFactory(1)).getPersonalTrainerDao();
    evaluationDao = Objects.requireNonNull(DaoFactory.getDaoFactory(1)).getEvaluationDao();
}
```

Figure 32: Il metodo *@BeforeAll* della classe di test *CostumerControllerTest*

Ugualmente importante è il metodo che permette la pulizia del "Database" prima dell'esecuzione di ogni metodo di test. Questo viene fatto dal momento che la sequenza con la quale vengono effettuati i test è casuale e cambia ad ogni lancio del test. Ogni metodo di test cambia i dati mantenuti dai dao (aggiunge un dato, ne rimuove altri, etc...) quindi per evitare che i confronti eseguiti con le liste mantenute dai dao siano veritieri, la pulizia del "database" prima del test risulta fondamentale.

```

@BeforeEach
void beforeEach(){
    trialSubscriptionDao.deleteAll();
    subscriptionDao.deleteAll();
    dailyDao.deleteAll();
    costumerDao.deleteAll();
    trainingCardDao.deleteAll();
    personalTrainerDao.deleteAll();
    evaluationDao.deleteAll();
}

```

Figure 33: Il metodo *@BeforeEach* della classe di test *CostumerControllerTest*

### 4.1.3 GymManagerControllerTest

Questa classe esegue il test dei metodi contenuti in *GymManagerController*. Questa classe presenta in maggioranza metodi relativamente semplici di aggiunta/rimozione di elementi dal database (receptionist e personal trainer) e metodi di visualizzazione. Come esempio viene analizzato il testing del metodo *getAllAccessFromDate()*, che ha come obiettivo quello di restituire una lista contenenti tutti gli accessi alla struttura in un determinato giorno selezionato dal gestore della palestra:

```

@Test
void getAllAccessFromDate() {
    LocalDateTime actualDateTime = LocalDateTime.now();

    Costumer costumer1 = new Costumer( name: "Prova", surname: "Test", phoneNumber: "863715361");
    Costumer costumer2 = new Costumer( name: "Ludovico", surname: "Siciliani", phoneNumber: "976425842");

    Access access1 = new Access(costumer1, actualDateTime);
    Access access2 = new Access(costumer2, actualDateTime);
    Access access3 = new Access(costumer2, actualDateTime.plusDays(2));
    Access access4 = new Access(costumer2, actualDateTime.plusDays(1));

    accessDao.add(access1);
    accessDao.add(access2);
    accessDao.add(access3);
    accessDao.add(access4);

    assertEquals( expected: 2,
        gymManagerController.getAllAccessFromDate(actualDateTime.getYear(),
            actualDateTime.getMonthValue(), actualDateTime.getDayOfMonth()).size()
    );
    assertEquals(access1,
        gymManagerController.getAllAccessFromDate(actualDateTime.getYear(),
            actualDateTime.getMonthValue(), actualDateTime.getDayOfMonth()).get(0)
    );
    assertEquals(access2,
        gymManagerController.getAllAccessFromDate(
            actualDateTime.getYear(),
            actualDateTime.getMonthValue(), actualDateTime.getDayOfMonth()).get(1)
    );
}

```

Figure 34: Il metodo *getAllAccessFromDate()*

#### 4.1.4 PersonalTrainerControllerTest

Questa classe testa *PersonalTrainerController*. Fra i metodi testati ha particolare rilevanza il metodo usato per l'aggiunta di una scheda specifica per un utente. Il seguente metodo quindi testa l'omonimo metodo della classe sotto test.

```
@Test
void addCustomizeTrainingCard() {
    PersonalTrainer personalTrainer = new PersonalTrainer( name: "Tommaso",
        surname: "Botarelli",
        phoneNumber: "7576143571"
    );

    personalTrainerDao.add(personalTrainer);

    Costumer costumer = new Costumer( name: "Sandro", surname: "Giusti", phoneNumber: "7862345");

    personalTrainerController.setThisPersonalTrainer(personalTrainer);

    personalTrainerController.addCustomizeTrainingCard(
        costumer, exercises: "Exercise", level: 1,
        emissionDay: 22, emissionMonth: 4, emissionYear: 2022, expirationDay: 22, expirationMonth: 5, expirationYear: 2022,
        name: "TrainingCard"
    );

    assertFalse(trainingCardDao.getTrainingCardFromCostumer(costumer).isEmpty());
}
```

Figure 35: Il metodo *addCustomizeTrainingCard()*

Il metodo imita che il personal trainer sia nel dao e che abbia eseguito l'accesso al sistema.

Un altro metodo interessante è quello che permette data una scheda di allenamento di averne una copia da assegnare ad un nuovo cliente.

```
void copyTrainingCard(){
    PersonalTrainer personalTrainer = new PersonalTrainer( name: "Tommaso", surname: "Botarelli", phoneNumber: "7576143571");

    personalTrainerDao.add(personalTrainer);

    personalTrainerController.setThisPersonalTrainer(personalTrainer);

    TrainingCard trainingCard = new TrainingCard(
        exercises: "Some exercises",
        level: 1,
        standard: true,
        personalTrainer
    );

    Costumer costumer1 = new Costumer( name: "Sandro", surname: "Giusti", phoneNumber: "7862345");

    personalTrainerController.copyTrainingCard(
        trainingCard,
        costumer1,
        LocalDate.now(),
        LocalDate.now().plusMonths(1),
        name: "CopiedTrainingCard"
    );

    TrainingCard trainingCardCopied = new TrainingCard( exercises: "Some exercises", level: 1, standard: false, personalTrainer);
    trainingCardCopied.setName("CopiedTrainingCard");
    trainingCardCopied.setEmission(LocalDate.now());
    trainingCardCopied.setExpiration(LocalDate.now().plusMonths(1));
    trainingCardCopied.setCostumer(costumer1);

    assertEquals(trainingCardCopied, trainingCardDao.getTrainingCardFromCostumer(costumer1).get(0));
}
```

Figure 36: Il metodo *copyTrainingCard*

Anche in questo caso viene imitato che il personal trainer sia presente nel dao e che abbia eseguito l'accesso al sistema.

#### 4.1.5 ReceptionistControllerTest

In questa classe viene testata la classe *ReceptionistController*. All'interno di questa classe grande importanza la ricopre il metodo *addAccessType()* utilizzato per aggiungere un tipo di abbonamento ad un determinato cliente. Il test del metodo appare così:

```
@Test
void addAccessType() {
    Costumer costumer1 = new Costumer( name: "Tommaso", surname: "Botarelli", phoneNumber: "8926735");
    costumerDao.add(costumer1);

    Bill genericBill = new Bill( payment: 20f, LocalDateTime.now());
    Long genericBillID = billDao.add(genericBill);

    receptionistController.addAccessType( type: "subscription", subscriptionType: "trial", genericBillID, LocalDateTime.now(), costumer1);

    assertEquals(LocalDate.now(), trialSubscriptionDao.getAll().get(0).getEmission());
    assertEquals(LocalDate.now().plusDays(14), trialSubscriptionDao.getAll().get(0).getExpiration());
    assertEquals(costumer1, trialSubscriptionDao.getAll().get(0).getCostumerTarget());

    receptionistController.addAccessType( type: "subscription", subscriptionType: "monthly", genericBillID, LocalDateTime.now(), costumer1);

    assertEquals(LocalDate.now(), subscriptionDao.getAll().get(0).getEmission());
    assertEquals(LocalDate.now().plusMonths(1), subscriptionDao.getAll().get(0).getExpiration());
    assertEquals(costumer1, subscriptionDao.getAll().get(0).getMyCostumer());
    assertEquals(TypeOfSub.MONTHLY, subscriptionDao.getAll().get(0).getTypeOfSub());

    subscriptionDao.deleteAll();

    receptionistController.addAccessType( type: "subscription", subscriptionType: "quarterly", genericBillID, LocalDateTime.now(), costumer1);

    assertEquals(LocalDate.now(), subscriptionDao.getAll().get(0).getEmission());
    assertEquals(LocalDate.now().plusMonths(3), subscriptionDao.getAll().get(0).getExpiration());
    assertEquals(costumer1, subscriptionDao.getAll().get(0).getMyCostumer());
    assertEquals(TypeOfSub.QUARTERLY, subscriptionDao.getAll().get(0).getTypeOfSub());

    subscriptionDao.deleteAll();

    receptionistController.addAccessType( type: "subscription", subscriptionType: "halfyearly", genericBillID, LocalDateTime.now(), costumer1);

    assertEquals(LocalDate.now(), subscriptionDao.getAll().get(0).getEmission());
    assertEquals(LocalDate.now().plusMonths(6), subscriptionDao.getAll().get(0).getExpiration());
    assertEquals(costumer1, subscriptionDao.getAll().get(0).getMyCostumer());
    assertEquals(TypeOfSub.HALFYEARELY, subscriptionDao.getAll().get(0).getTypeOfSub());

    subscriptionDao.deleteAll();

    subscriptionDao.deleteAll();

    receptionistController.addAccessType( type: "subscription", subscriptionType: "yearly", genericBillID, LocalDateTime.now(), costumer1);

    assertEquals(LocalDate.now(), subscriptionDao.getAll().get(0).getEmission());
    assertEquals(LocalDate.now().plusMonths(12), subscriptionDao.getAll().get(0).getExpiration());
    assertEquals(costumer1, subscriptionDao.getAll().get(0).getMyCostumer());
    assertEquals(TypeOfSub.YEARLY, subscriptionDao.getAll().get(0).getTypeOfSub());

    subscriptionDao.deleteAll();

    receptionistController.addAccessType( type: "daily", subscriptionType: "", genericBillID, LocalDateTime.now(), costumer1);

    assertEquals(LocalDate.now(), dailyDao.getAll().get(0).getEmission());
    assertEquals(costumer1, dailyDao.getAll().get(0).getMyCostumer());
}
```

Figure 37: Il test del metodo *addAccessType()*

## 4.2 Black box testing

In questa classe di Test viene effettuata una "simulazione" del cliente che fa accesso alla struttura e che usufruisce dei servizi, il personal trainer e il receptionist di conseguenza

aggiungono rispettivamente le schede di allenamento e l'abbonamento.

Di seguito alcune parti della classe in questione più importanti:

```
/*
First of all the personal trainer do a first evaluation of the new customer.
*/

personalTrainerWindow.addEvaluation(
    personalTrainerWindow.selectCustomer( name: "Marco", surname: "De Luca", phoneNumber: "3456789087"),
    LocalDate.now().getYear(),
    LocalDate.now().getMonthValue(),
    LocalDate.now().getDayOfMonth(),
    comments: "Some comments about measurements and evaluation",
    progressLevel: 2,
    height: 1.80f,
    weight: 70f,
    leanMass: 62,
    fatMass: 8
);

/*
The personal trainer set a default training card (progress level 2) to the customer.
*/

personalTrainerWindow.copyTrainingCard(
    personalTrainerWindow.getMyDefaultTrainingCard( progressLevel: 2),
    personalTrainerWindow.selectCustomer( name: "Marco", surname: "De Luca", phoneNumber: "3456789087"),
    LocalDate.now(),
    LocalDate.now().plusDays(14),
    name: "TrainingCard copied"
);

/*
The customer can see the training card from his window, first of all the customer login to the system.
*/

try {
    customerWindow = loginWindow.createCustomerSession( name: "Marco", surname: "De Luca", phoneNumber: "3456789087");
}
catch(Exception e){
    System.out.println(e.getMessage());
}

try {
    customerWindow.getMyCurrentTrainingCard(
        LocalDate.now().getDayOfMonth(),
        LocalDate.now().getMonthValue(),
        LocalDate.now().getYear()
    );
}
catch(Exception e){
    /*
    If the system recognize that the customer has no trainingCard show an error dialog.
    */
    System.out.println(e.getMessage());
}
```

```

/*
Then with the badge want to access to the gym for the 4th time but the system show an error dialog to the
receptionist, because he can't access to the gym.
*/

```

```

try{
    receptionistWindow.addAccessForCustomerFromBadge( id: 0, LocalDateTime.now().plusDays(12));
}
catch(Exception e){
    System.out.println("PRIMA STAMPA: " + e.getMessage());
}

```

```

/*
The receptionist receive the payment and a new subscription for the customer is added to the system.
*/

```

```

receptionistWindow.addAccessType( type: "subscription",
    subscriptionType: "monthly",
    receptionistWindow.addPayment(TypeOfSub.MONTHLY, LocalDateTime.now().plusMonths(1)),
    LocalDate.now().plusMonths(1),
    receptionistWindow.selectCustomer( name: "Marco", surname: "De Luca", phoneNumber: "3456789087"));

```

```

/*
The receptionist receive the payment and a new subscription for the customer is added to the system.
*/

```

```

receptionistWindow.addAccessType( type: "subscription",
    subscriptionType: "monthly",
    receptionistWindow.addPayment(TypeOfSub.MONTHLY, LocalDateTime.now().plusMonths(1)),
    LocalDate.now().plusMonths(1),
    receptionistWindow.selectCustomer( name: "Marco", surname: "De Luca", phoneNumber: "3456789087"));

```

```

/*
The customer use the badge for the access. (suppose that the sensor recognize the badge id)
*/

```

```

try {
    receptionistWindow.addAccessForCustomerFromBadge( id: 0, LocalDateTime.now().plusMonths(1));
} catch (Exception e) {
    System.out.println(e.getMessage());
}

```

```

/*
The personal trainer add a new customize training card for the customer.
*/

```