Relazione progetto ‘ManaGYMent’ per programmazione ad oggetti

Davide Borficchia, Simone Letizi, Federico Giannoni

31 Maggio 2015

Indice

1 Analisi

1.1 Requisiti 3
1.2 Problema.....4

2 Design

2.1 Architettura 5

2.2 Design dettagliato

2.2.1 Divisione package 7

2.2.2 Modello dettagliato 9

2.2.3 Pattern utilizzati 16

3 Sviluppo

3.1 Testing automatizzato 22

3.2 Divisione dei compiti e metodologia di lavoro 22

3.3 Note di sviluppo

3.3.1 Scelte critiche 23

3.3.2 Fonti e materiale attinto 24

3.3.3 Scelte logico-implementative 24

4 Commenti finali

4.1 Conclusioni e lavori futuri 26

A Guida utente 27

Capitolo 1

Analisi

1.1 Requisiti

Il programma ManaGYMent ha lo scopo di gestire ed organizzare la parte amministrativa e gestionale di una palestra.
L'applicazione è creata e pensata per essere utilizzata da uno o più utenti, siano essi il proprietario e/o gestore della palestra.
Nello specifico, ManaGYMent, nasce per soddisfare le seguenti funzionalità:

- Gestione multiutente
- Gestione del programma settimanale della palestra dove l'amministratore organizza per fascia oraria i corsi offerti dalla palestra
- Gestione di iscritti tramite una finestra per aggiungerli, modificarli ed eliminarli dalla palestra.
- Gestione del personale tramite una finestra per aggiungerlo, modificarlo ed eliminarlo dalla palestra.
- Gestione del simil-bilancio mensile calcolato sulla differenza tra gli stipendi pagati al personale e i crediti riscossi dagli abbonamenti degli iscritti
- Gestione dei corsi interni alla palestra, con la possibilità di aggiungere/rimuovere uno o più insegnanti
- Possibilità di inviare notifiche e news agli impiegati e/o iscritti

1.2 Problema

Come già annunciato, ManaGYMent dovrà essere in grado di garantire la multiutenza applicativae la persistenza dei dati dei vari utilizzatori salvando e caricando i dati interessati ad ogni chiusura e ad ogni apertura dell'applicativo.
Naturalmente dovrà essere garantito il funzionamento multipiattaforma dato che le palestre potrebbero usare differenti sistemi operativi o macchine.

Anche la gestione del programma settimanale rappresenta una problematica di importanza, in quanto sarà necessario escogitare un metodo per rappresentarlo graficamente nella maniera più chiara ed intuitiva possibile.

Un'ulteriore difficoltà è rappresentata inoltre dall'implementazione della funzionalità di gestione dei membri della palestra (personale e iscritti) in quanto entità i cui dati variano con lo scorrere del tempo (basti pensare alla scadenza dell'abbonamento di un iscritto o al pagamento mensile del salario di un impiegato).

Infine, dal momento che le entità che verranno manipolate dall'applicazione sono tra loro strettamente collegate, le modifiche che verranno fatte su una di esse avranno molto probabilmente delle forti ripercussioni anche sulle altre.
Sarà quindi necessario individuare il modo più adatto per gestire tali ripercussioni (ad esempio: *Come dovrà comportarsi l'applicazione a seguito della cancellazione di un impiegato assegnato ad un corso già presente nel programma settimanale*?).

Capitolo 2

Design

2.1 Architettura

Il pattern architetturale su cui si basa il cuore della nostra applicazione è l'MVC (Model – View - Controller) per garantire la netta separazione fra i diversi aspetti di presentazione e visualizzazione, di rappresentazione dei dati utilizzati, e di aspetti legati al funzionamento e alla logica dell'applicativo.

Nella figura 2.1 viene mostrato lo scheletro della struttura principale dell'applicazione.
Nel suddetto schema la parte composta da GenericPanel, GenericPanelController e le rispettive interfacce è stata inserita esclusivamente nell'UML (non è presente a livello di codice) per evidenziare come una qualunque vista possa attaccarsi al 'core' architetturale del programma.
Notare come le varie viste possano essere attaccate all'architettura principale senza intervenire su di essa (sarà sufficiente adattare il controller personale della vista a quest'ultima).
Per quanto riguarda le viste e i rispettivi controller nello specifico questi verranno mostrati nel dettaglio nel paragrafo **2.2 Design dettagliato**, assieme al modello.

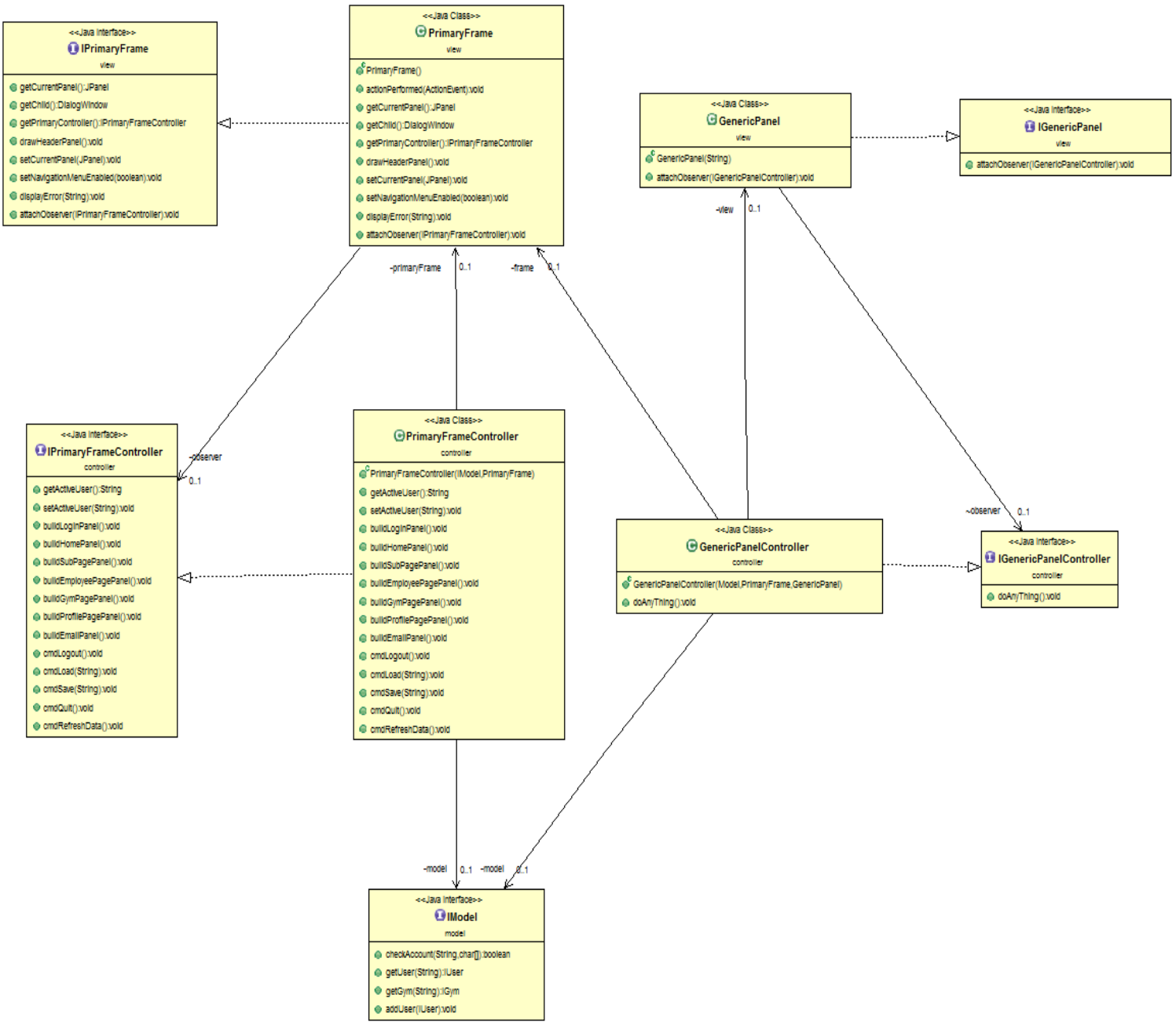

<<Java Interface>>
IPrimaryFrame
view
getCurrentPanel():JPanel
getChild():DialogWindow
getPrimaryController():IPrimaryFrameController
drawHeaderPanel():void
setCurrentPanel(JPanel):void
setNavigationMenuEnabled(boolean):void
displayError(String):void
attachObserver(IPrimaryFrameController):void
<<Java Class>>
PrimaryFrame
view
PrimaryFrame()
actionPerformed(ActionEvent):void
getCurrentPanel():JPanel
getChild():DialogWindow
getPrimaryController():IPrimaryFrameController
drawHeaderPanel():void
setCurrentPanel(JPanel):void
setNavigationMenuEnabled(boolean):void
displayError(String):void
attachObserver(IPrimaryFrameController):void
<<Java Class>>
GenericPanel
view
GenericPanel(String)
attachObserver(IGenericPanelController):void
<<Java Interface>>
IGenericPanel
view
attachObserver(IGenericPanelController):void
-primaryFrame
0..1
-frame
0..1
-view
0..1
<<Java Interface>>
IPrimaryFrameController
controller
getActiveUser():String
setActiveUser(String):void
buildLogInPanel():void
buildHomePanel():void
buildSubPagePanel():void
buildEmployeePagePanel():void
buildGymPagePanel():void
buildProfilePagePanel():void
buildEmailPanel():void
cmdLogout():void
cmdLoad(String):void
cmdSave(String):void
cmdQuit():void
cmdRefreshData():void
-observer
0..1
<<Java Class>>
PrimaryFrameController
controller
PrimaryFrameController(IModel,PrimaryFrame)
getActiveUser():String
setActiveUser(String):void
buildLogInPanel():void
buildHomePanel():void
buildSubPagePanel():void
buildEmployeePagePanel():void
buildGymPagePanel():void
buildProfilePagePanel():void
buildEmailPanel():void
cmdLogout():void
cmdLoad(String):void
cmdSave(String):void
cmdQuit():void
cmdRefreshData():void
<<Java Class>>
GenericPanelController
controller
GenericPanelController(IModel,PrimaryFrame,GenericPanel)
doAnyThing():void
-observer
0..1
<<Java Interface>>
IGenericPanelController
controller
doAnyThing():void
-model
0..1
-model
0..1
<<Java Interface>>
IModel
model
checkAccount(String,char[]):boolean
getUser(String):IUser
getGym(String):IGym
addUser(IUser):void

Figura 2.1

2.2 Design dettagliato

2.2.1 Divisione in Package

Per quanto riguarda la divisione in package, abbiamo deciso di effettuarla in questo modo:

- **controller:** contiene l'interfaccia e la classe del controller relativo al frame principale
- **controller.panels.email:** contiene le interfacce e le classi dei controller relative al pannello utilizzatoper l'invio delle email
- **controller.panels.gym:** contiene le interfacce e leclassi dei controller relativi al pannello della palestra (bilancio e corsi offerti) e al dialog aggiungere/modificare un corso
- **controller.panels.home**: contiene le interfacce e le classi dei controller relativi al pannello della home (calendario settimanale) e al dialog per settare il programma giornaliero
- **controller.panels.login:** contiene le interfacce e le classi dei controller relativi al pannello della home (calendario settimanale) e al dialog per settare il programma giornaliero
- **controller.panels.members**:contiene le interfacce e le classi dei controller relativi ai pannelli di aggiunta e modifica di impiegati e iscritti
- **controller.panels.members.tables:** contiene l'interfaccia per la classe astratta dei controller relativi alla visualizzazione dei membri(impiegati e iscritti), la classe astratta e le specifiche classi che la estendono
- **controller.panels.profile**: contiene l'interfaccia e la classe del controller relativo al pannello del profilo (modifica password e modifica email)
- **controller.panels.signup**: contiene l'interfaccia e la classe del controller relativo al pannello di registrazione
- **main:** contiene solo la classe main, che si occupa dell'avvio dell'applicazione sulla pagina di Login
- **model:** contiene le classi riguardanti il modello e gli utenti dell' applicazione (Model e User),con le rispettive interfacce
- **model.gym:** contiene le classi riguardanti la palestra (Gym, Schedule, Course, GymCalendar) con le rispettive interfacce
- **model.gym.members:** contiene le classi riguardanti i membri della palestra (Subscriber,Employee), con le relative interfacce e con la classe astratta che racchiude le parti comuni
- **utility:** contiene la classe UtilityClass, composta di soli metodi statici
- **view:** contiene l' interfaccia e la relativa classe della vista incaricata di mostrare il frame principale

- **view.panels.email**: contiene l'interfaccia e la classe della vista incaricata di mostrare il pannello per inviare delle email
- **view.panels.gym:** contiene le interfacce e le classi delle viste incaricate di visualizzare il calendario e il settaggio di quest'ultimo.
- **view.panels.home:** contiene le interfacce e le classi delle viste incaricate di visualizzare la pagina della palestra, (che mostra il bilancio mese per mese e i corsi offerti) e la pagina relativa all'aggiunta/modifica di un corso
- **view.panels.login:** contiene l' interfaccia e la relativa classe della vista incaricata di mostrare la pagina di registrazione
- **view.panels.members:** contiene le interfacce e le classi delle viste incaricate di visualizzare le pagine di aggiunta/modifica di iscritti e impiegati
- **view.panels.members.tables:** contiene l' interfaccia e la classe della vista incaricata di mostrare iscritti e impiegati della palestra tramite una tabella
- **view.panels.profile:** contiene l' interfaccia e la classe della vista incaricata di mostrare la pagina relativa alla modifica del profilo loggato
- **view.panels.signup:** contiene l' interfaccia e la classe della vista incaricata di mostrare la pagina di registrazione

2.2.2 Modello dettagliato

Per cominciare la figura 2.2 mostra l'UML della parte di modello relativa agli utenti.

Come è possibile vedere dall'UML il modello può contenere più utenti , ciascuno dei quali associato ad una sola palestra.

La struttura di questa palestra è evidenziata nella figura 2.3 e nella figura 2.4

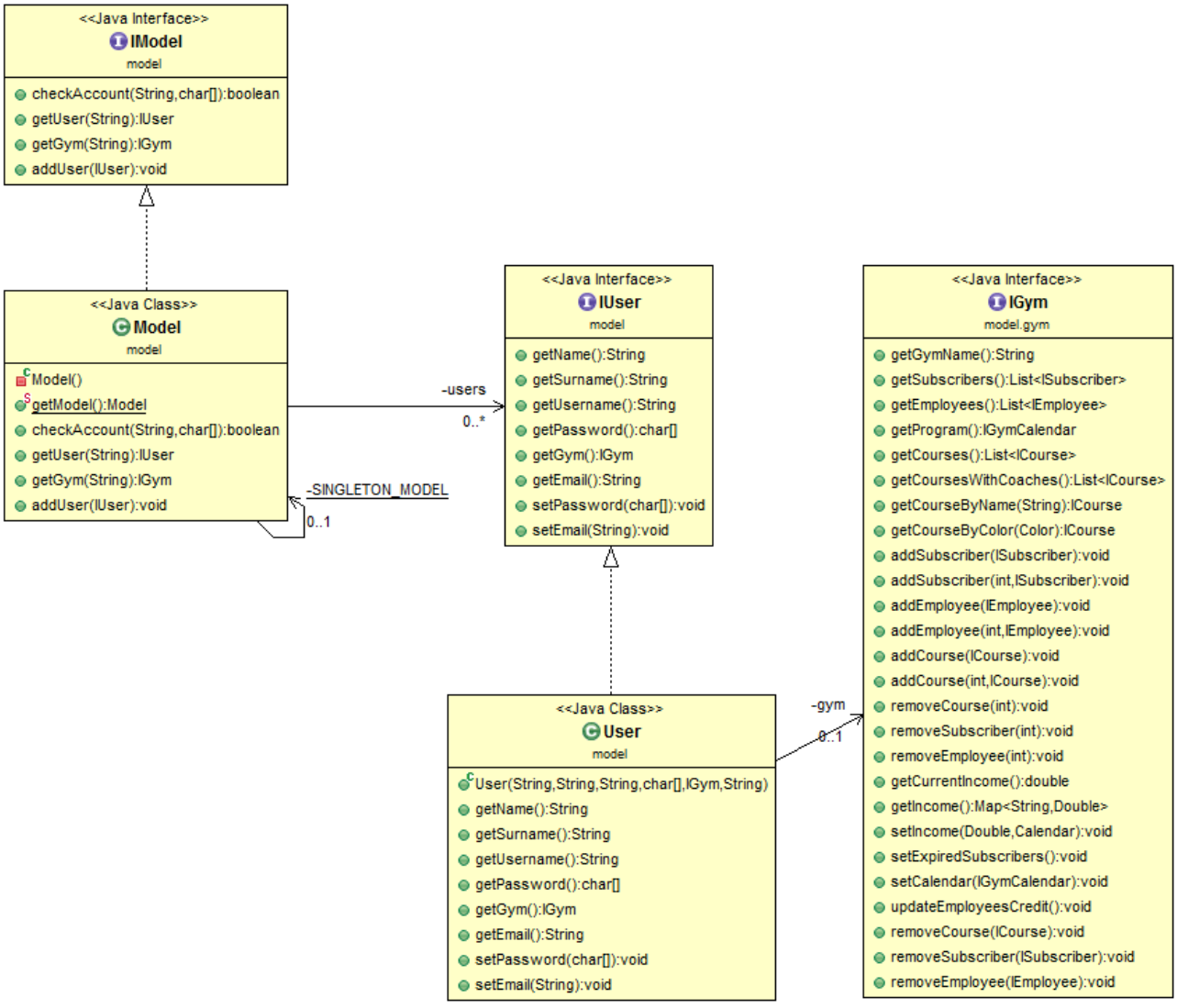


Figura 2.2

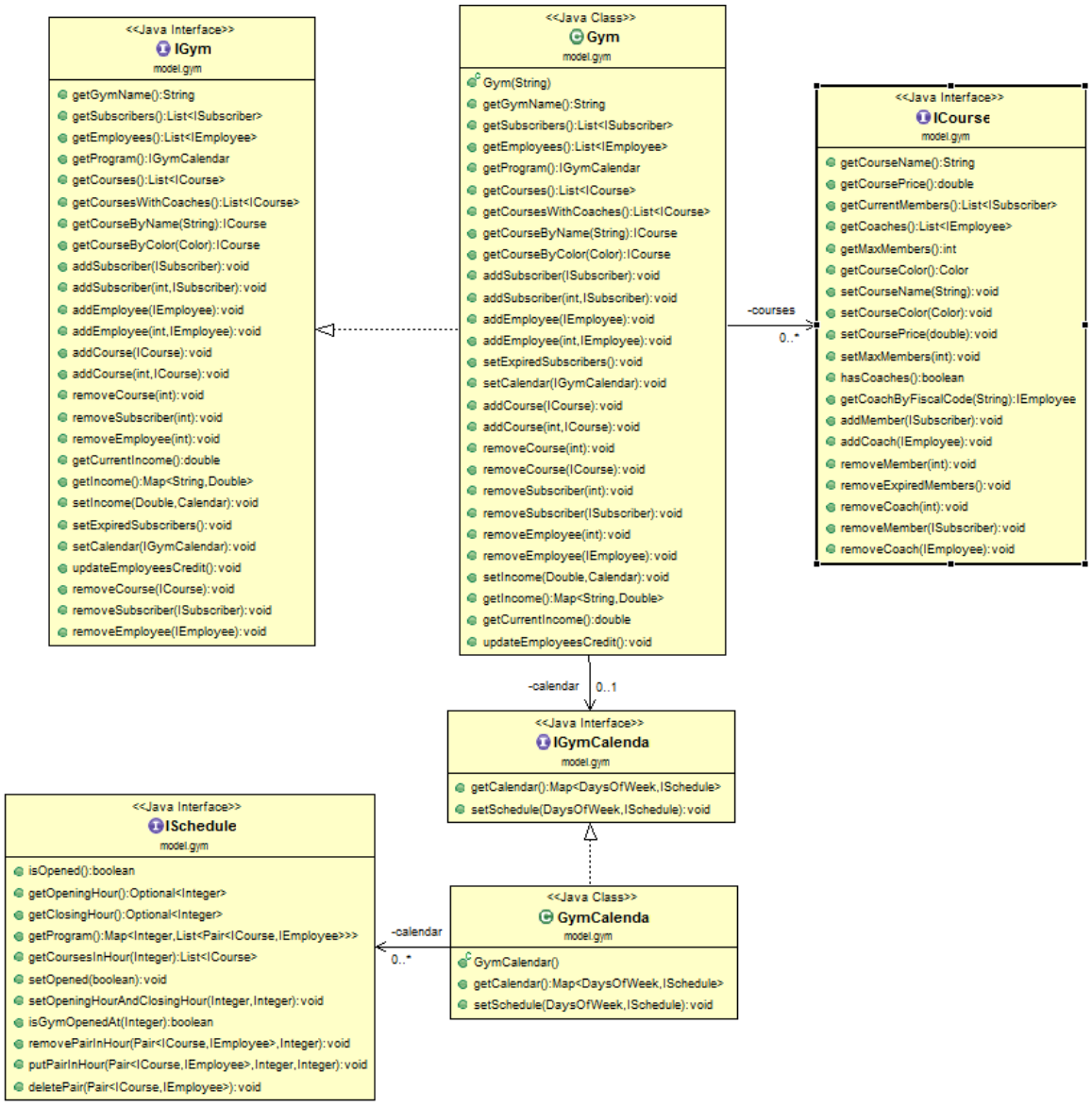


Figura 2.3: Schema UML della palestra di un utente (sono esclusi i membri della palestra, che verranno evidenziati in uno schema successivo). Sono state omesse dallo schema alcune classi (Course e Schedule) per non rendere lo schema troppo caotico.

Questo schema mette in evidenza l'aspetto organizzativo della palestra, cioè come è possibile organizzare il programma della palestra.

Una palestra contiene un calendario settimanale (IGymCalendar) che è composto da un programma giornaliero (ISchedule) per ognuno dei sette giorni della settimana. La palestra inoltre, contiene un numero indeterminato di corsi, che possono essere accoppiati (tramite una classe Pair generica) a degli insegnanti ed in seguito aggiunti ad una Schedule del calendario (IGymCalendar). In questo modo è possibile pianificare, tramite l'applicazione, il programma settimanale che la palestra offre.

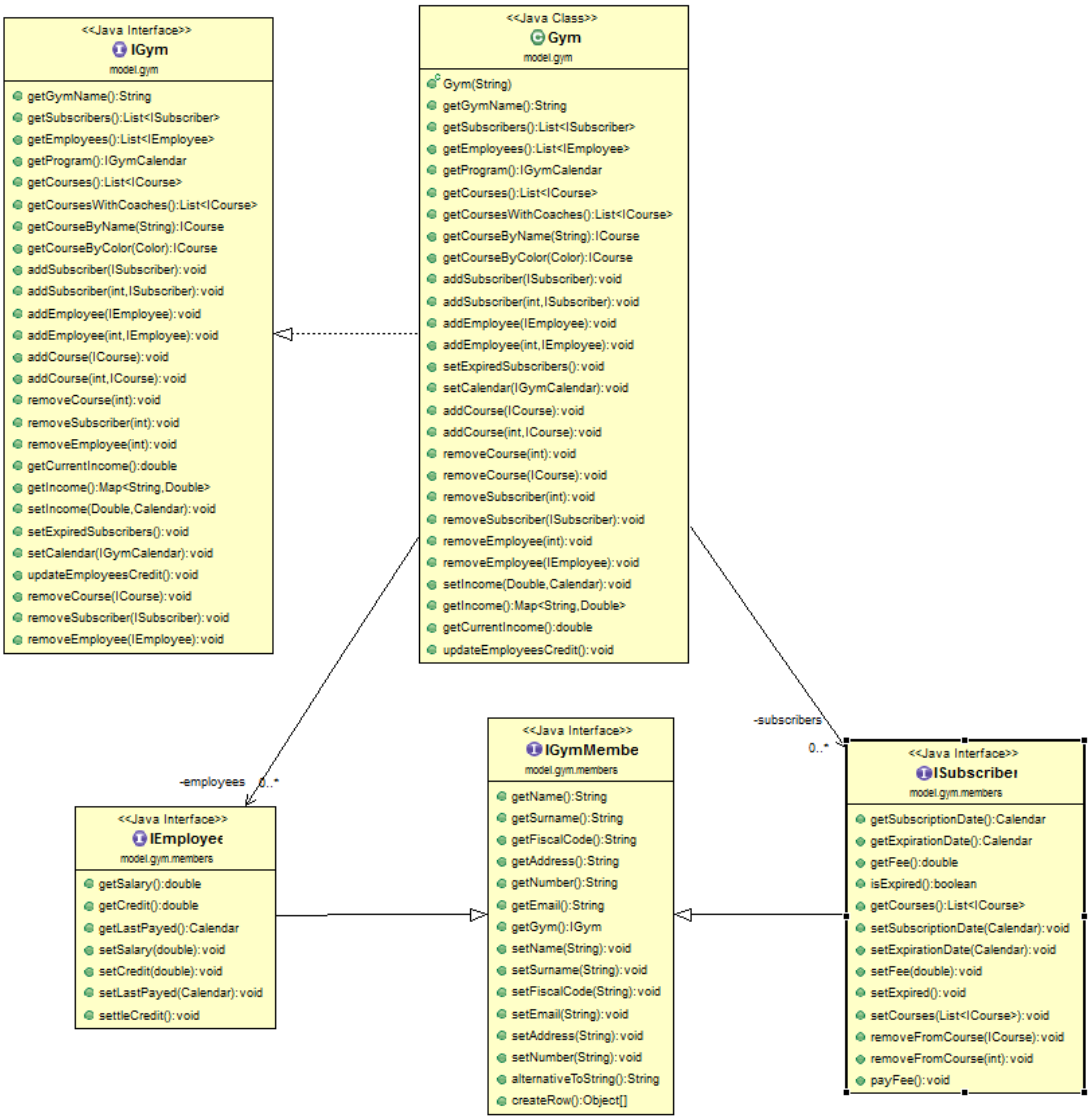


Figura 2.4: Questa figura mostra invece l'UML relativo alla parte membri: iscritti ed impiegati. Ogni palestra può contenere un numero indeterminato di entrambi. L'interfaccia IGymMember racchiude le parti di contratto comuni alle due classi ed è implementata da una classe astratta (non rappresentata nell'UML per problemi di spazio).

Un altro dettaglio non presente nello schema (sempre per motivi di spazio), infatti, riguarda il fatto che in Subscriber è presente una ridondanza. Infatti, benchè l'entità Course, rappresentata nella figura 2.3, contenga una lista di iscritti, anche ognuno degli iscritti, contiene la lista dei corsi di cui è membro (questa scelta è stata effettuata per velocizzare alcune operazioni). Il giorno in cui l'abbonamento di un iscritto scade, esso viene anche rimosso da tutti i corsi a cui era iscritto (per lasciare spazio ad altri iscritti, in quanto i corsi hanno un numero massimo di membri).

Vediamo ora le parti relative alle varie viste e i rispettivi controller. Per alcuni di questi, si è fatto uso di finestre di dialogo (le quali hanno portato a non pochi problemi implementativi e alla necessità di utilizzare delle copie temporanee di parti di modello. Questi problemi saranno approfonditi nel capitolo **3.3 Note di sviluppo**).

Vediamo gli UML delle viste principali e dei rispettivi controller.
Si è stati attenti a non duplicare codice, utilizzando ad esempio delle tabelle astratte in comune per rappresentare sia l'elenco degli impiegati che degli iscritti. Inoltre, nelle viste utilizzate per l'inserimento / modifica di dati di utenti e membri della palestra (es. pannello per la registrazione, pannello per l'aggiunta/modifica di membri), è stato applicato uno Strategy per fare in modo che tali viste siano più facilmente mantenibili nel caso di modifiche di campi del modello.

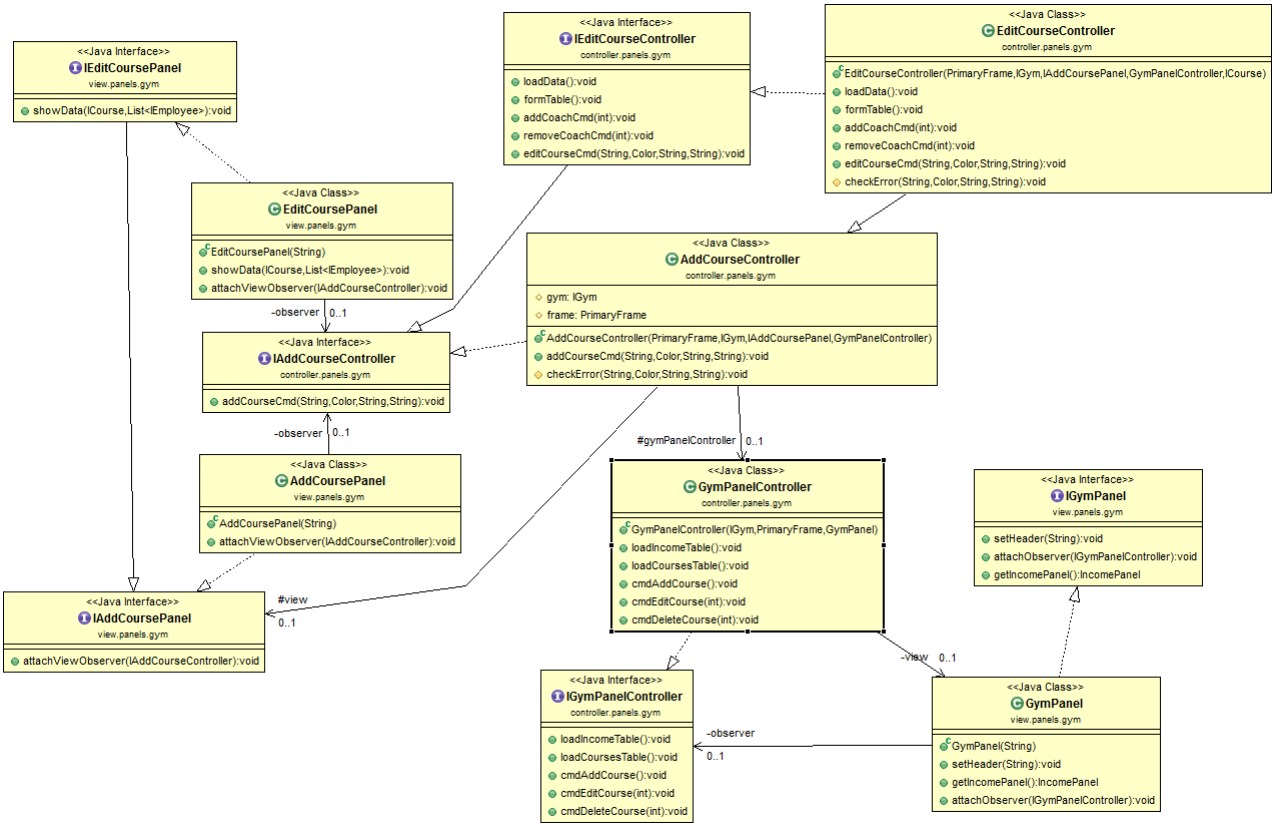


Figura 2.5: UML della parte di vista e relativi controller utilizzati per visualizzare / aggiungere / modificare / eliminare i corsi di una palestra. Questa parte dell'applicazione (EditCourseController) permette anche di aggiungere insegnanti ai vari corsi (gli insegnanti vengono presi dalla lista di impiegati della palestra). Viene anche mostrato il bilancio della palestra per ogni mese a partire da quello di iscrizione.

Per quanto riguarda la modalità di aggiunta e modifica dei corsi, dovendo quest'ultima anche fornire la possibilità di aggiungere insegnanti, si è deciso di far estendere a EditCourseController la classe AddCourseController per gestire la verifica di eventuali errori di inserimento(stringhe al posto di interi, nome del corso già presente ..) tramite il metodo protectedcheckError, essendo i controlli uguali sia per l'aggiunta che per la modifica ed eccezionedel caso in cui il numero di membri massimi del corso inserito sia inferiore al numero già presente in lista(per questo il metodo viene ridefinito in EditCourseController con una piccola aggiunta).

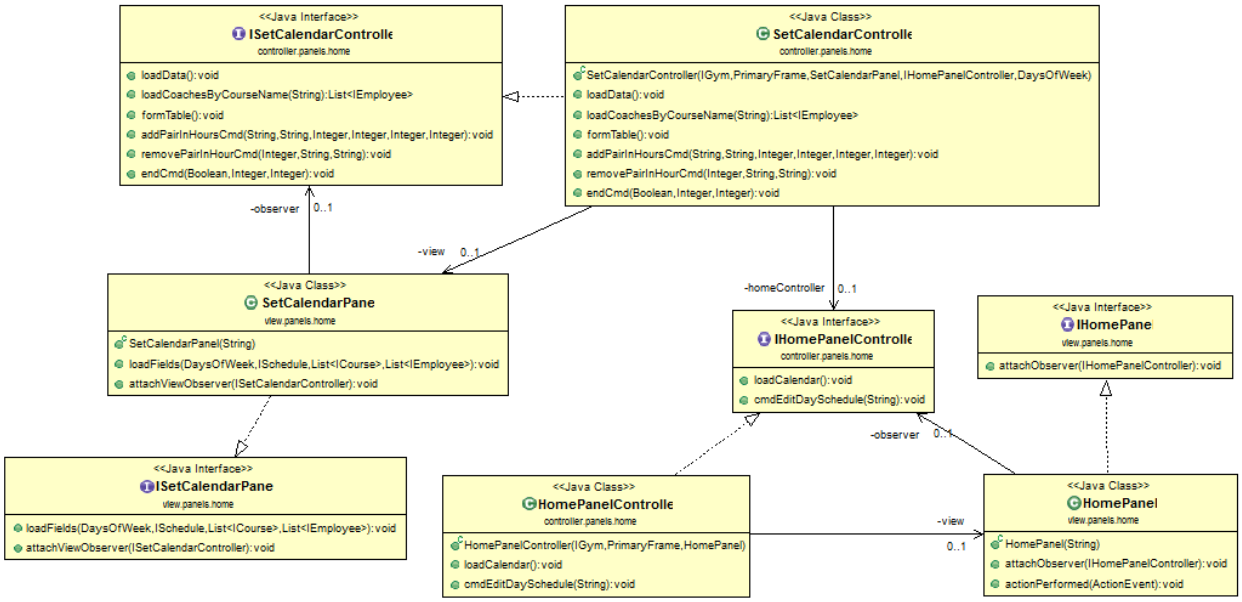


Figura 2.6: UML della parte di vista e controller relativi alla home page dell'applicazione.
In questa parte è possibile organizzare il programma settimanale della palestra, visualizzato tramite HomePanel.
Per la maggior parte delle view principali si è deciso di mostrare dei dialogaggiuntivi a comparsa al posto di ridisegnare sempre lo stesso frame.
Si noti infatti come in SetCalendarController sia presente IHomePanelController, per permettere, una volta impostato il programma, di effettuare il cambiamento del calendario tramite il metodo HomePanelController.loadCalendar(), per poi chiudere il dialog.

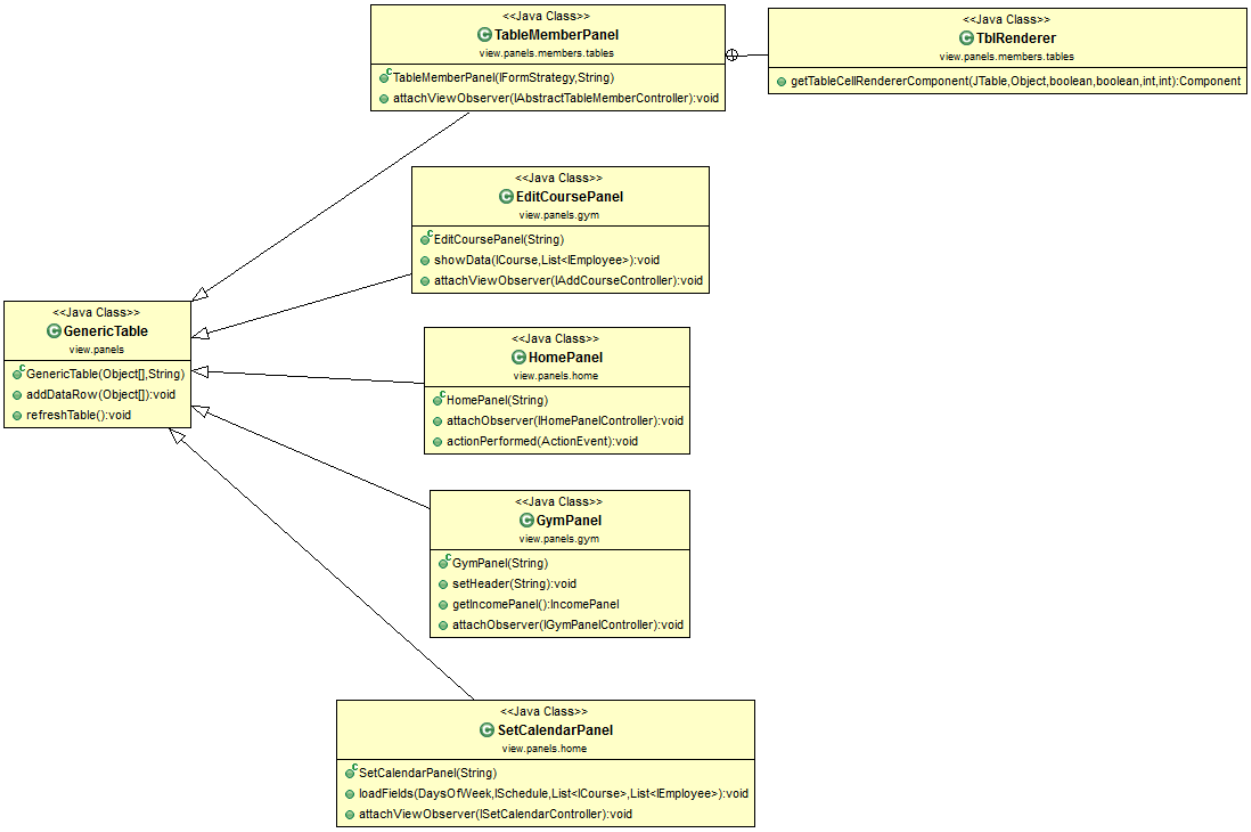


Figura 2.7: la figura sopra rappresentata vuole mostrare come l'uso delle JTable sia molto frequente nel nostro programma, tanto da spingerci a creare una classe GenericTable che racchiuda i comportamenti comuni delle nostre tabelle (aggiunta di una righa nella tabella e l'intero refresh di essa) in modo da essere estesa dagli altri pannelli che facessero uso di JTable semplicemente richiamando il costruttore della superclassspecificando gli header.

2.2.3 Pattern Utilizzati

Singleton:

Per garantire la creazione di una sola istanza di Model e di PrimaryFrame, con accesso globale a tale istanza si è fatto uso del pattern creazionale singleton.
Nella figura 2.2 di pagina 9 possiamo notare come il costruttore di Model sia private per impedire l'istanziazione diretta della classe.
grazie al metodo getModel() statico viene ritornata l'unica istanza possibile di tale classe.
Stesso procedimento è stato utilizzato per PrimaryFrame.

Observer:

Cogliamo l'occasione per mostrare una parte del programma non ancora vista e per presentare come viene applicato il pattern observer su di essa.

Come già detto ogni pannello implementa una propria interfaccia, ogni interfaccia possiede un metodo (attachObserver) che prende come parametro l'interfaccia dell'observer e non fa altro che assegnarla al campo observer presente dentro ogni pannello.

Saranno poi i vari listeners ad assegnare le varie operazioni degli observer agli eventi della GUI, oppure saranno i controller ad inviare alla GUI informazioni da gestire.

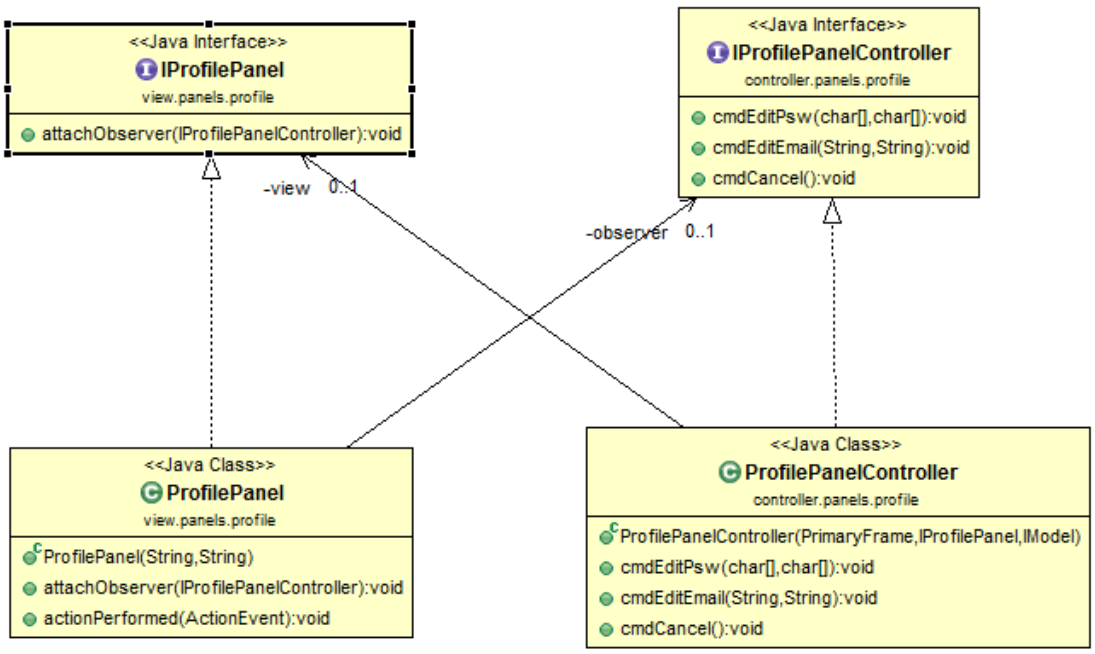
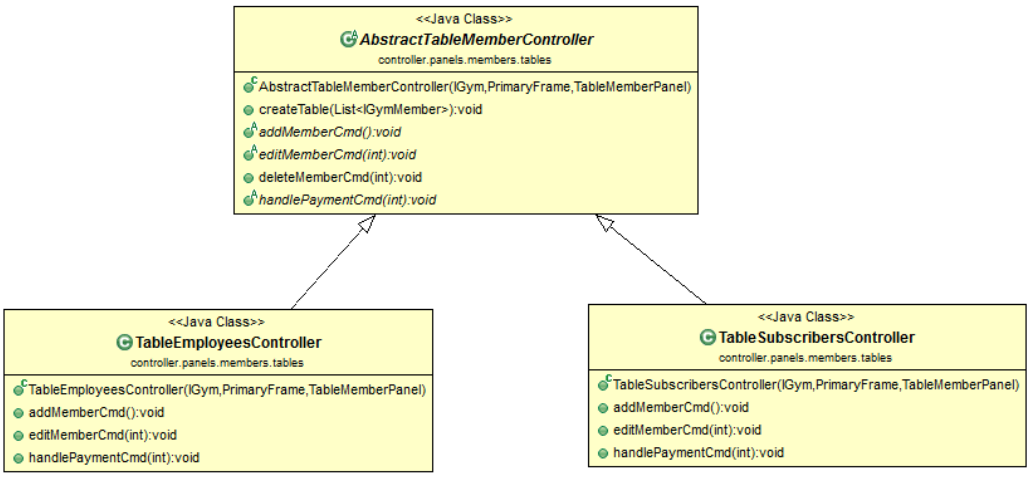


Figura 2.8: Questo UML rappresenta l'applicazione del pattern observer per quanto riguarda il pannello del profilo (modifica della mail e della password di registrazione).

Si noti come la classe ProfilePanel contenga l'interfaccia IProfilePanelController e il metodo attachObserver() per assegnare il campo di quest'ultimo e per effettuare determinati comandi, a seconda degli eventi della GUI che si verificano.

Template method:



come si è voluto mostrare precedentemente le classi employee e subscriber sono molto simili e anche per quanto riguarda il controller relativo alle tabelle su cui mostrarli si è deciso di duplicare meno codice possibile implementando una classe astratta che racchiudesse i metodi comuni(come ad esempio il caricamento di questi nella tabella) .

```java
public abstract class AbstractTableMemberController implements IAbstractTableMemberController {

    private static final String CONFIRM = "Sei sicuro di voler cancellare il membro selezionato?";
    protected final PrimaryFrame frame;
    protected final IGym gym;
    private final TableMemberPanel view;

    /**
     * @param gym
     *            the gym
     * @param frame
     *            the application's frame
     * @param view
     *            the view
     */
    public AbstractTableMemberController(final IGym gym, final PrimaryFrame frame, final TableMemberPanel view) {
        this.gym = gym;
        this.frame = frame;
        this.view = view;
        this.view.attachViewObserver(this);
    }

    public void createTable(final List<? extends IGymMember> list) {
        this.view.refreshTable();
        list.forEach(member -> {
            this.view.addDataRow(member.createRow());
        });
    }

    public abstract void addMemberCmd();

    public abstract void editMemberCmd(final int index);

    protected abstract void deleteMember(final int index);

    public void deleteMemberCmd(final int index) {
        final int n = JOptionPane.showConfirmDialog(this.view, CONFIRM, "Conferma", JOptionPane.YES_NO_OPTION);
        if (n == JOptionPane.YES_OPTION) {
            this.deleteMember(index);
        }
    }

    public abstract void handlePaymentCmd(final int index);
```

Figura 2.9: Nella figura si può notare l'uso del templatemethod.
infatti, il metodo deleteMemberCmd(), che si occupa della visualizzazione del messaggio di conferma è uguale sia per gli impiegati che per gli iscritti, ma al suo interno richiama deletemember(), implementato diversamente in ognuna delle sottoclassi, che rimuove effettivamente il membro dalla lista presente nel modello

Strategy:

l'uso del pattern comportamentale strategy si è prestato utile sia per i controller delle viste che si occupano dell'aggiunta e la modifica di membri generici (per il controllo dell'inserimento esatto dei dati tramite il metodo getpred() in IFormField) sia per la creazione della tabella di visualizzazione di essi (per determinare gli header e il nome degli option button relativi ai filtri, differenti in alcune parti da membro a membro). L'implementazione scelta è basata sulle ENUM, in modo che i campi possano essere aggiunti con molta facilità.

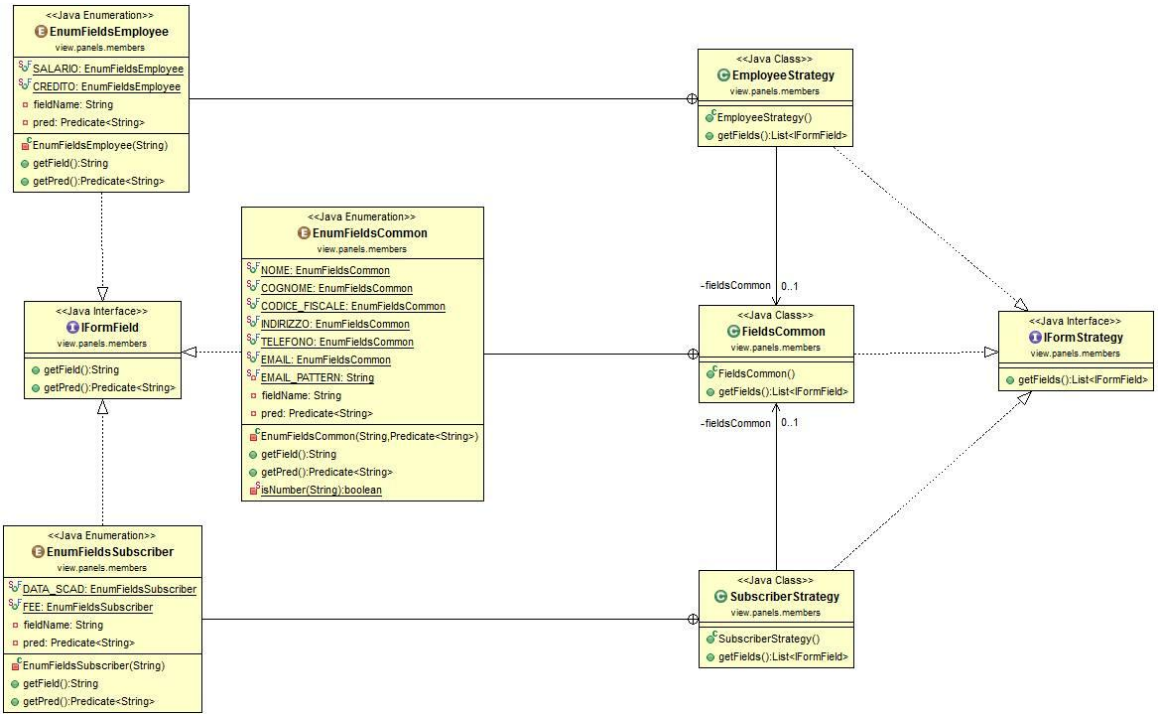


figura 2.10: nella precedente figura si mostra la strategy IFormStrategy (interfaccia invocata dal contesto in cui è usata in base all'algoritmo scelto) e le sue specificazioni.
Si noti come il metodo getFields (che ritorna i campi di un membro) sia replicato in ognuna delle sottoclassi e implementato da queste in maniera diversa in base ai campi (iformfield) che ogni membro possiede, ritornandoli sotto forma di lista .

inoltre ISubscriberStrategy e IEmployeeStrategy estendano FieldsCommon, ridefinendo getfields(), che non fa altro che richiamare il getfields() della superclasse e aggiungerci i campi specifici.

ogni strategy contiene al suo interno una ENUM, che implementa IFormField; il compito di ogni enum è quello di memorizzare, per ogni campo il nome di esso e la condizione di validità del campo stesso (es: il numero di telefono deve essere per forza un numero e non una stringa, il CF deve essere lungo 15 caratteri).

nelle prossime figure verrà mostrato il contesto in cui queste strategy vengono utilizzate (purtroppo per problemi di spazio e di leggibilità non è stato possibile mostrare tutto in un UML unico).

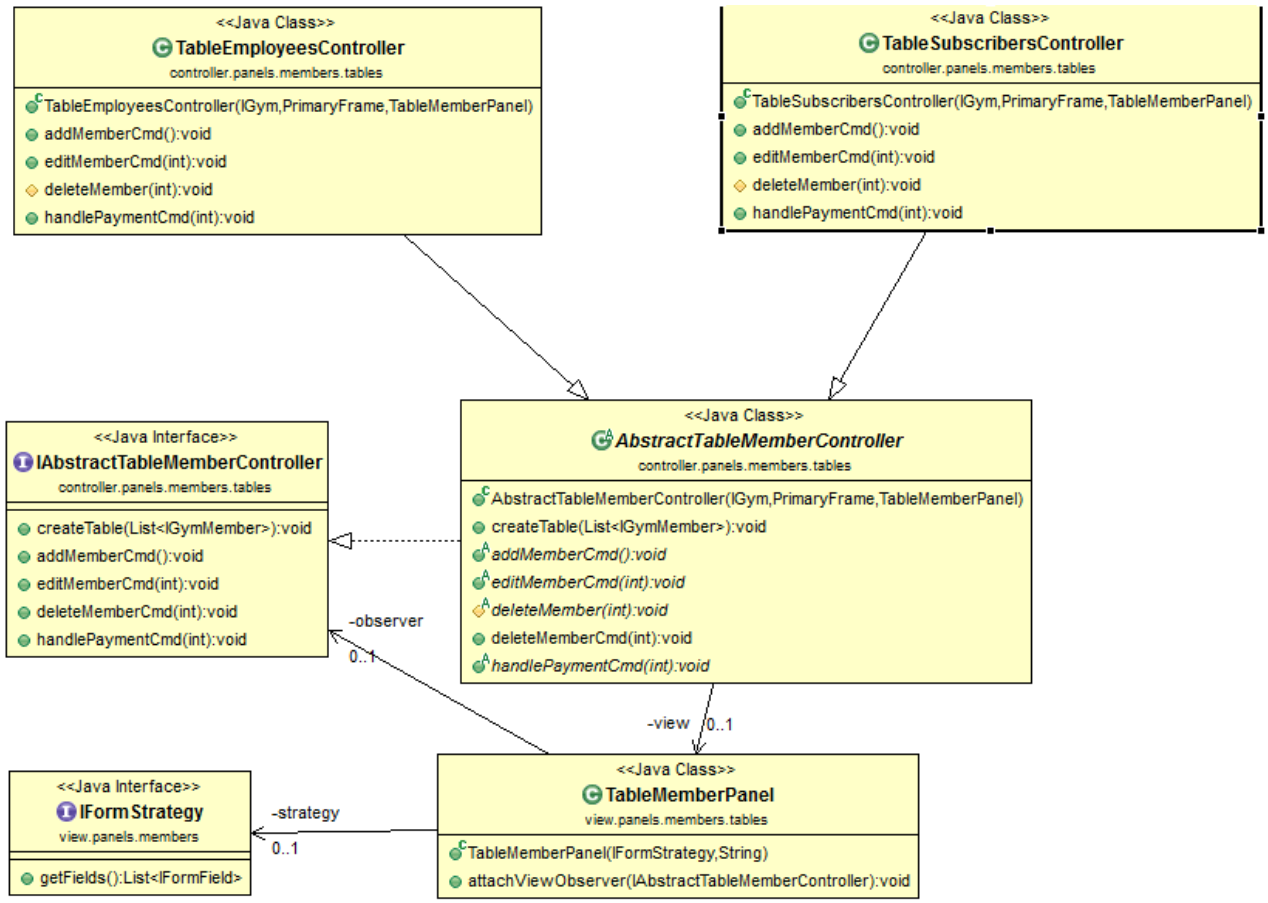


figura 2.11

la figura sopra rappresentata vuole evidenziare il contesto in cui la strategy viene usata. guardando il costruttore di TableMemberPanel ci si può accorgere che uno degli argomenti è appunto IFormStrategy, usato per determinare, come già sottolineato gli header della tabella e il testo sopra i componenti della view.

Si è sfruttata questa figura anche per mostrare l'interazione tra view e controller di questa parte, in modo da mostrare nel complesso l'astrazione del controller per i membri citata precedentemente.

Nella figura 2.12 infine, è mostrata un'ulteriore applicazione del pattern strategy, questa volta relativa all'inserimento di un nuovo utente.

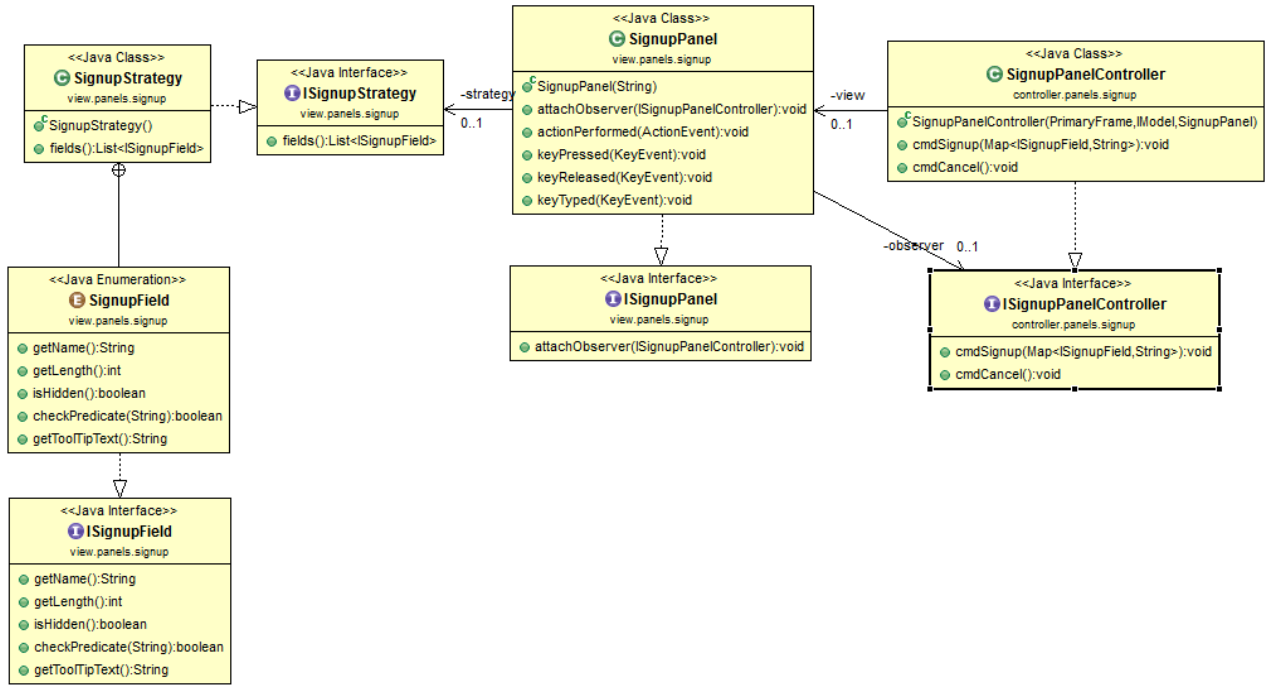


Figura 2.12:

Anche in questo caso, è presente un enumeratore contenente i vari campi necessari per la registrazione di un nuovo utente ed un predicato che stabilisce la condizione di validità da rispettare per ogni campo. All'inserimento dei campi richiesti per la creazione di un nuovo utente, gli input verranno mappati al valore dell'enumeratore corrispondente. Il metodo checkPredicate() dell'enumeratore controllerà poi se il valore inserito per quel campo supera o meno la condizione di validità di quel campo.
Il nuovo utente verrà creato solo se tutti i valori dell'enumeratore (ottenibili mediante il metodo fields() in SignupStrategy) superano le condizioni di validità dei rispettivi campi.

L'utilizzo di questo pattern, oltre a garantire la separazione tra le componenti di vista e modello, facilita anche operazioni di manutenzione sulla parte di vista nel caso in cui il modello dovesse subire delle modifiche (ad es. se cambiano i campi richiesti per la registrazione di un utente, sarà sufficiente semplicemente operare sull'enumeratore aggiungendo/rimuovendo/modificandone i campi e i rispettivi predicati di validità).

Capitolo 3
Sviluppo

3.1 Testing automatizzato

Per testare il funzionamento dell'applicazione, sono stati svolti dei test automatizzati con JUnit. Una volta garantito il corretto funzionamento della parte di modello, i vari controller sono stati invece testati manualmente, in quanto il lavoro è stato diviso in modo che ogni componente del gruppo lavorasse su parti di modello e su parti di vista e controller relative a tali parti di modello.

Una volta garantito il funzionamento delle parti svolte individualmente, queste sono state unite dall'intero gruppo.

La fase di testing è stata poi ripresa una volta unite le varie parti in maniera individuale. Ogni volta che un membro del gruppo individuava un bug o un comportamento non corretto dell'applicazione, esso procedeva con il fixing, dopo essersi consultato con il resto del gruppo (è quindi capitato che alcuni membri del gruppo intervenissero sul codice di altri membri, ma sempre sotto la supervisione del resto del gruppo).

3.2 Divisione dei compiti e metodologia di lavoro

Il lavoro è stato diviso tra i 3 componenti del gruppo come segue:

- Giannoni Federico: Realizzazione dello scheletro del modello (che è stato poi espanso da tutti i membri del gruppo) e delle principali strutture dati da utilizzare in esso. Realizzazione delle view e dei relativi controller per la parte di registrazione, per il frame principale, per la visualizzazione del calendario settimanale(Home Page), per la pagina della palestra (bilancio e corsi disponibili), per la gestione del bilancio
- Letizi Simone: Realizzazione delle view e dei relativi controller per la visualizzazione di iscritti e di impiegati, per la parte relativa all'aggiunta e alla modifica di un corso, per la parte relativa al settaggio del programma giornaliero

- Borficchia Davide: Realizzazione delle view e dei relativi controller per la parte correlata all'inserimento e alla modifica di iscritti e impiegati, per quella del profilo(modifica password ed email) e per quella relativa all'invio delle email
- Insieme: Divisione package, sviluppo di tutto il modello, applicazione MVC.

Una volta creato il modello con i metodi principali il lavoro è stato diviso in modo che ogni componente del gruppo riuscisse a svolgere quasi autonomamente il lavoro assegnatogli, basato su controller e vista di alcune delle parti, cercando sempre di mantenere un rigore logico tra le parti assegnate.

Nonostante la semplicità nel lavorare autonomamente, essendo comunque un lavoro di gruppo è capitato frequentemente di incontrarsi per affrontare alcuni dei problemi elencati nel primo capitolo.

L'utilizzo di Mercurial è stato significativo dopo il primo mese di lavoro sul modello, in quanto prima si è lavorato localmente per definire lo scheletro del programma.

Come richiesto dal prof, abbiamo creato un repository e una volta ben definito il modello lo si è caricato in modo di lavorare tutti su quello.

Attraverso il plugin per eclipse è molto semplice effettuare commit, push, pull, merge e tutte le altre operazioni disponibili, in poco tempo.

3.3 Note di sviluppo

3.3.1 scelte critiche

Per rendere più pulita l'applicazione dal punto di vista grafico si è deciso di delegare le operazioni di aggiunta/modifica dei vari corsi, membri ecc. a delle finestre di dialogo.
Questo tuttavia, come già espresso ampliamente ha portato a delle complicazioni a livello di codice.
Il problema nasce perché le operazioni principali svolte su queste finestre comportano modifiche dirette sul modello.
Queste modifiche devono essere reversibili nel caso in cui l'utente chiuda la finestra senza aver confermato le modifiche fatte. (in tal caso bisognerebbe eliminare tutte le modifiche fatte).
Per fare ciò si è presa in considerazione la possibilità di usare il pattern memento; tuttavia l'utilizzo di esso avrebbe comportato l'aggiunta di numerose classi/interfacce e non avrebbe offerto vantaggi concreti, in quanto gli stati da ripristinare sarebbero stati uno solo e non più di uno.

Di conseguenza si è deciso di lavorare semplicemente su una copia dell'entità che si sarebbe dovuta modificare.
La necessità di modificare direttamente il modello è nata, oltre per una maggior chiarezza grafica (grazie all'uso delle jtable) anche per gestire le eccezioni mano a mano che le operazioni vengano eseguite.
Un controllo finale infatti sarebbe poco chiaro e molto dispendioso a livello di tempo in caso i corsi inseriti nelle varie ore fossero quasi tutti errati (nel caso del calendario ad esempio).
Le tabelle presenti in questi pannelli sono inoltre "refreshate"ad ogni operazione cancellando la tabella e ricreandola direttamente da modello

3.3.2 Fonti e materiale attinto

Trattandosi del nostro primo progetto e non essendo quindi sicuri di come procedere nel lavoro, il gruppo ha deciso di aiutarsi prendendo spunto dai vari progetti degli anni scorsi, osservando come questi ultimi fossero stati organizzati.

Anche l'implementazione di alcune parti del programma ha portato a diverse difficoltà, in particolare quelle relative all'utilizzo delle JTable, con le quali nessun membro del gruppo aveva particolare dimestichezza. Oltre allo studio della javadoc e dei tutorial Oracle, è risultato molto utile affidarsi anche ad altre risorse sul web, quali stack overflow, per risolvere dei dubbi che altrimenti sarebbero rimasti irrisolti.
Un esempio è offerto dal problema del costruire una tabella con dei bottoni cliccabili come headers.

3.3.3 Scelte logico-implementative

Come deducibile dagli schemi UML, le varie parti del modello relative alla palestra, sono tutte strettamente collegate tra loro, tanto che una in modifica in una di queste parti, può provocare delle ripercussioni sulle altre entità.

Per questo è stato necessario stabilire come l'applicazione debba comportarsi in questi casi.

Per quanto riguarda il calendario si è deciso di fare in modo che una palestra possa essereaperta in un determinato giorno, solo se sono previsti dei corsi per tale giorno.

Inoltre, è stato deciso, che la cancellazione di un impiegato, già designato come insegnante di alcuni specifici corsi, provochi anche la cancellazione dal programma delle ore nelle quali l'impiegato era in servizio (cioè vengono cancellate tutte le coppie impiegato da cancellare – corso insegnato, all'interno del programma settimanale della palestra). Inoltre, se a seguito di tale operazione, la palestra risultava essere aperta, ma senza corsi, in alcuni giorni, è stato deciso di fare in modo che essa venisse chiusa automaticamente (in altre parole, una palestra non può, come già detto, essere aperta senza alcun corso in programma).

Lo stesso principio è stato applicato per la rimozione di un corso già presente all'interno del programma settimanale della palestra. Inoltre, si è anche fatto in modo, che se venisse cancellato un corso con degli iscritti attivi (cioè non ancora scaduti), i giorni di abbonamento dei quali gli iscritti non potranno usufruire, vengano rimborsati.

Non è stata invece consentita l'operazione di rimozione di un insegnate di un corso già presente nel programma settimanale, in quanto tale operazione avrebbe avuto delle conseguenze sul modello impossibili da gestire.

Nella parte dei membri gli impiegati sono caratterizzati da un salario mensile e da un credito nei confronti della palestra, che viene incrementato una volta al mese (cioè, una volta al mese, il credito di un impiegato aumenta di una quantità pari al suo salario). Ogni volta che un impiegato viene pagato (operazione che va fatta manualmente), l'interno credito dell'impiegato viene azzerato e la transazione viene salvata sul bilancio relativo al mese corrente.

Per quanto riguarda gli iscritti invece, ognuno possiede informazioni relative al suo abbonamento (data di iscrizione, di scadenza, corsi a cui è iscritto). Ad ogni iscrizione, viene calcolata la somma che il membro dovrà versare alla palestra. Fino a quanto il membro non avrà saldato il suo debito (operazione che va fatta manualmente), non sarà possibile modificare la sua data di iscrizione/scadenza o i corsi a cui è iscritto.

Capitolo 4
Commenti finali

4.1 Conclusioni e lavori futuri

Nonostante il tanto impegno e la massima volontà nel curare sia l'aspetto grafico, che quello implementativo riteniamo che il progetto abbia alcune cose da migliorare.

Soprattutto nella gestione del bilancio ci sarebbero parecchie migliorie da fare.
Una di queste consiste nel gestire lo stipendio di un impiegato come un contratto vero e proprio e modellare il concetto di abbonamento come entità a sé stante, fornendo la possibilità di scegliere diversi tipi di esso (MENSILE, ASSOCIATO, CON CORSI BONUS E NON).
Un'altra miglioria sul bilancio, con il fine di renderlo più veritiero possibile, sarebbe quella di far inserire all'utente una cifra relativa ai costi fissi mensili, in modo conteggiare anche questi ultimi nel bilancio complessivo.

Appendice A
Guida utente

All'avvio dell'applicazione la prima pagina che verrà visualizzata è quella in figura 5.1, tramite il quale si può effettuare la registrazione o il login.

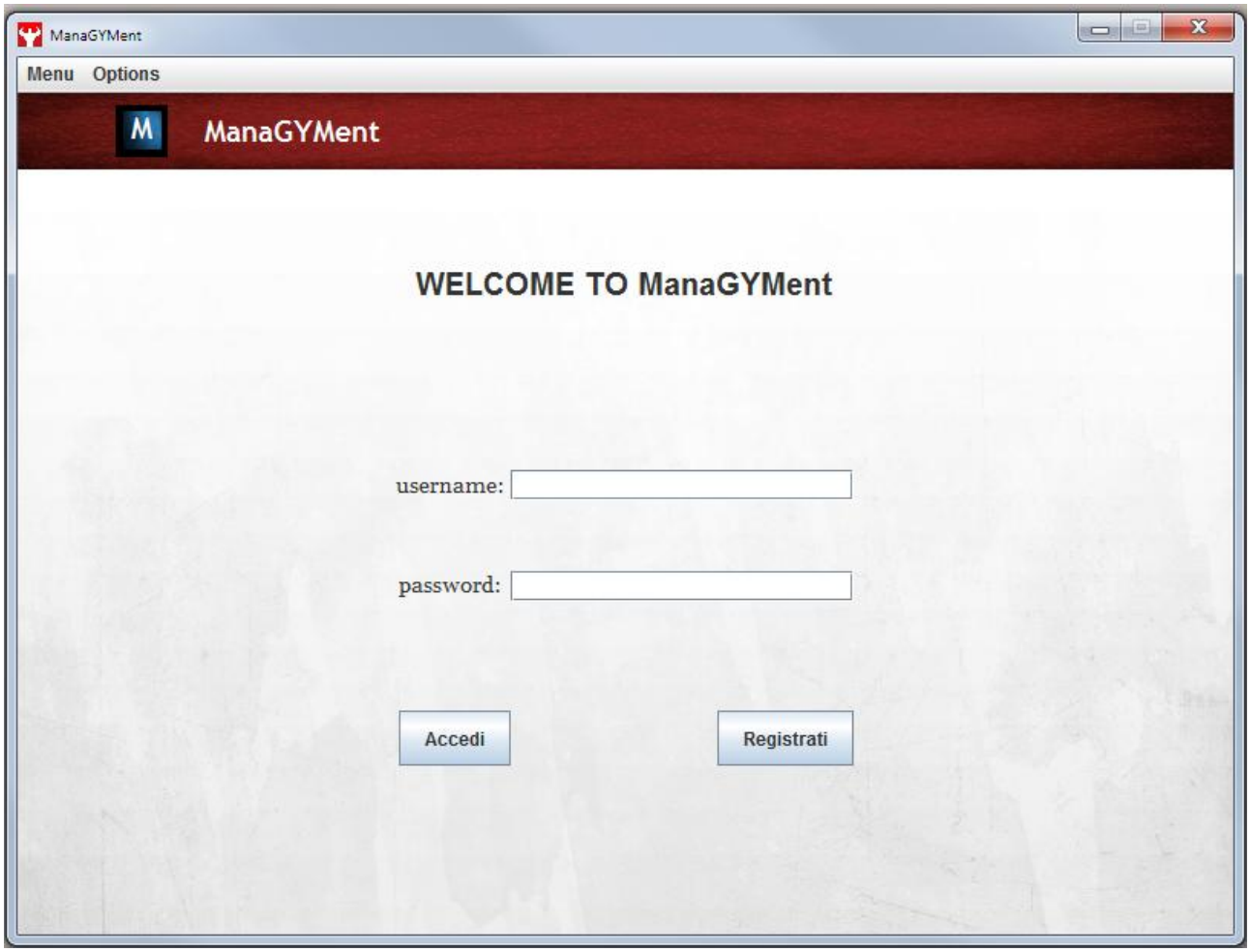
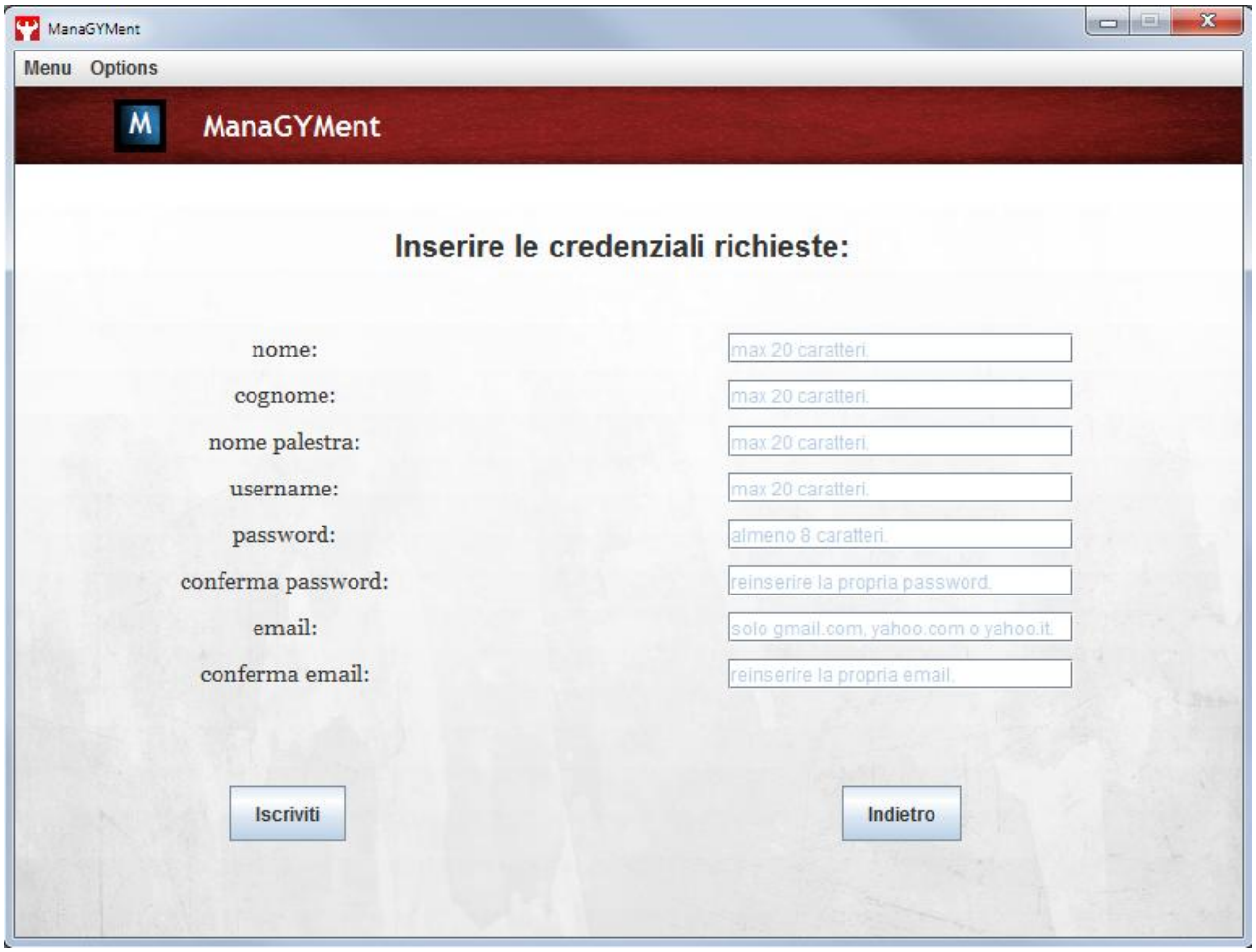


Figura 5.1

Cliccando su registrati si aprirà la pagina in figura 5.2, nel quale si devono inserire le informazioni per creare un nuovo account utente, mentre cliccando su accedi si accede direttamente all'applicazione inserendo i dati corretti relativi ad un account già esistente.



Una volta registratisi e inseriti username e password corretti la pagina di home è la seguente:

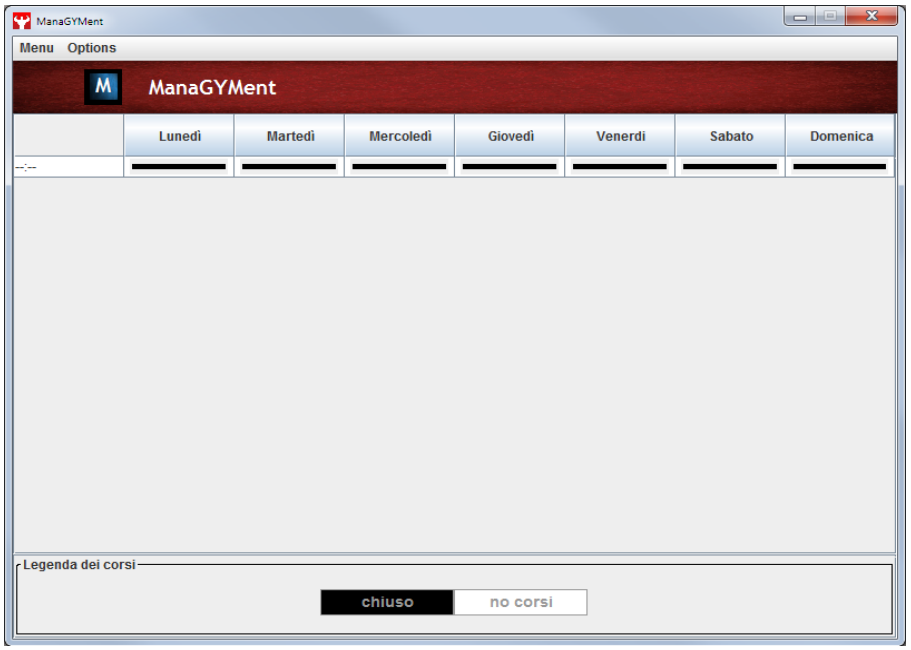


Figura 5.3

Siccome inizialmente la palestra sarà priva di corsi e di insegnanti si consiglia di inserirli aprendo il menu in alto a sinistra e cliccando su palestra. Si aprirà la schermata mostrata in figura 5.4 (anche se qui ci sono già corsi):

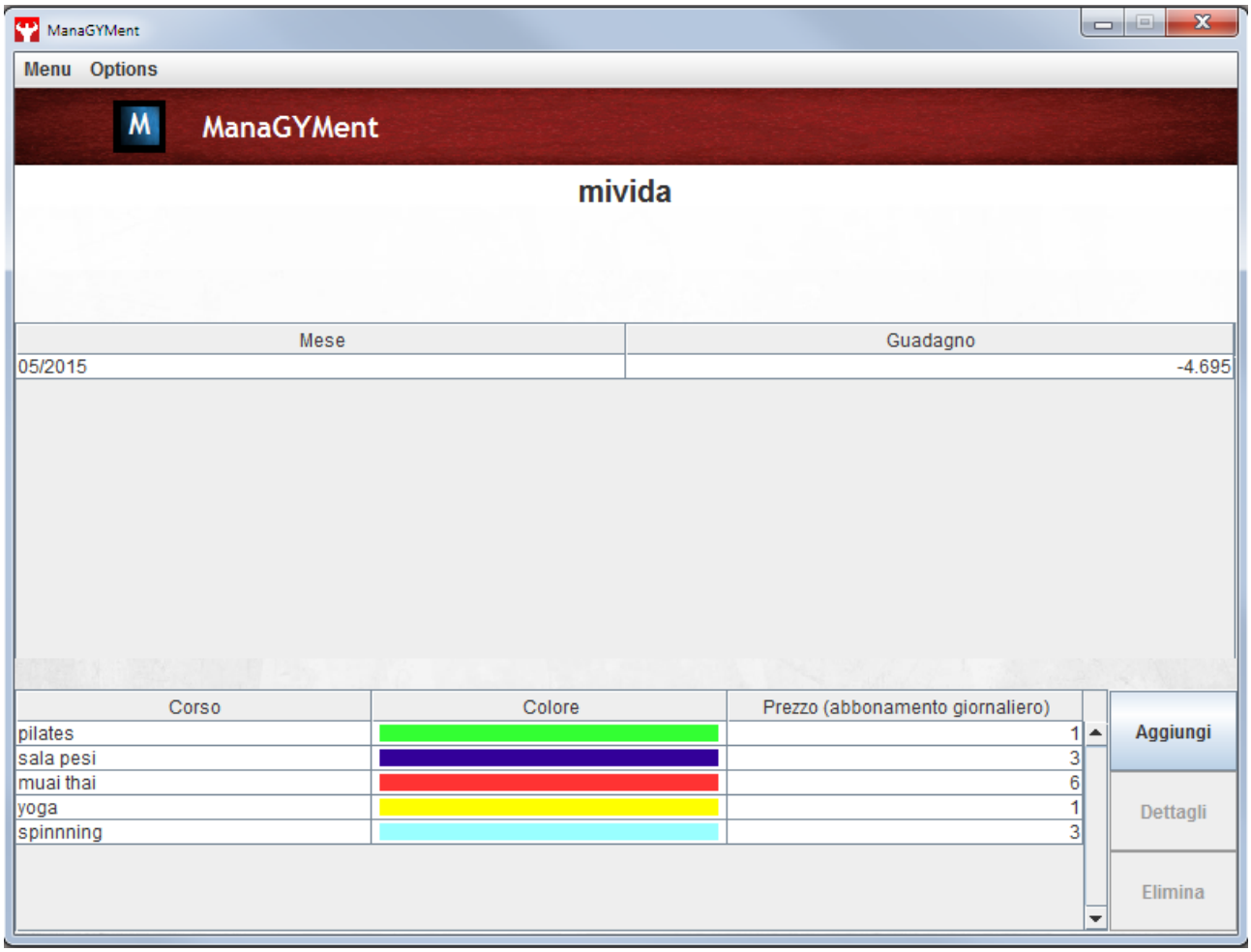


Figura 5.4

Una volta aggiunti i corsi procedere con l'inserimento degli impiegati cliccando IMPIEGATI nel menu a tendina.si aprirà una schermata come quella in figura 5.5, dove, oltre a visualizzare gli impiegati si da la possibilità di aggiungere, modificare, cancellare e saldare il conto di uno di essi.

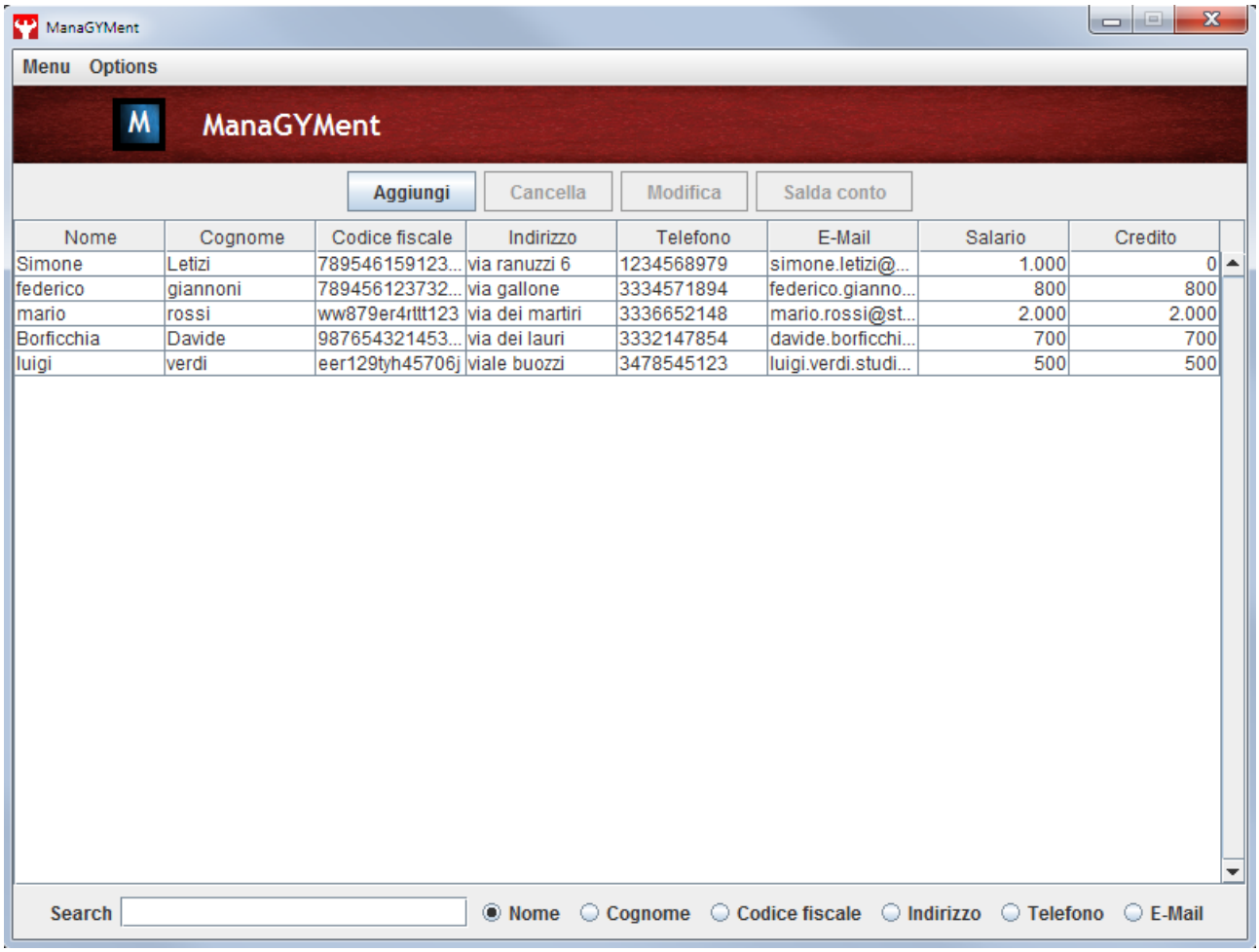
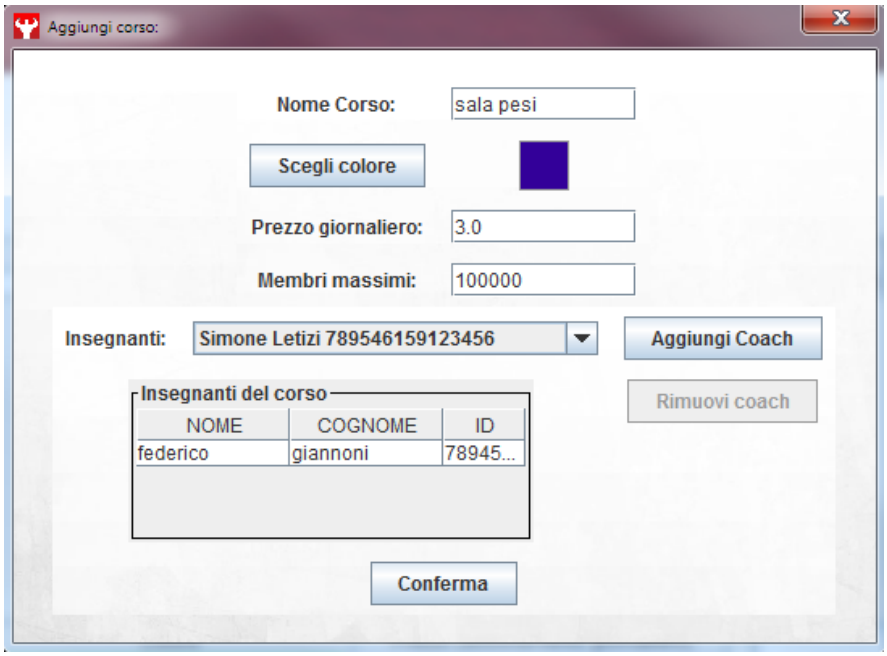
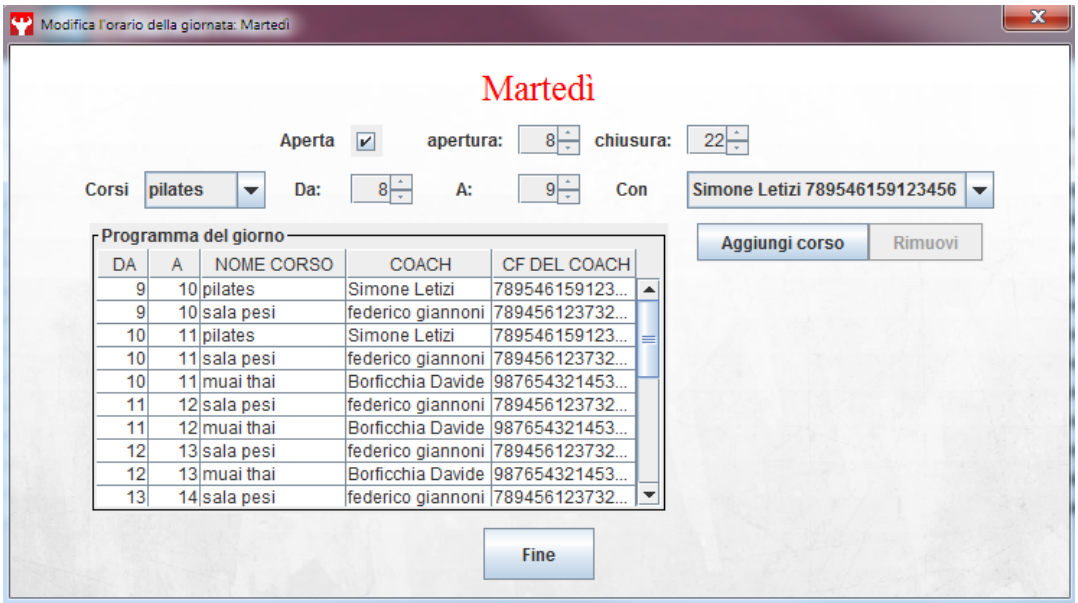


Figura 5.5

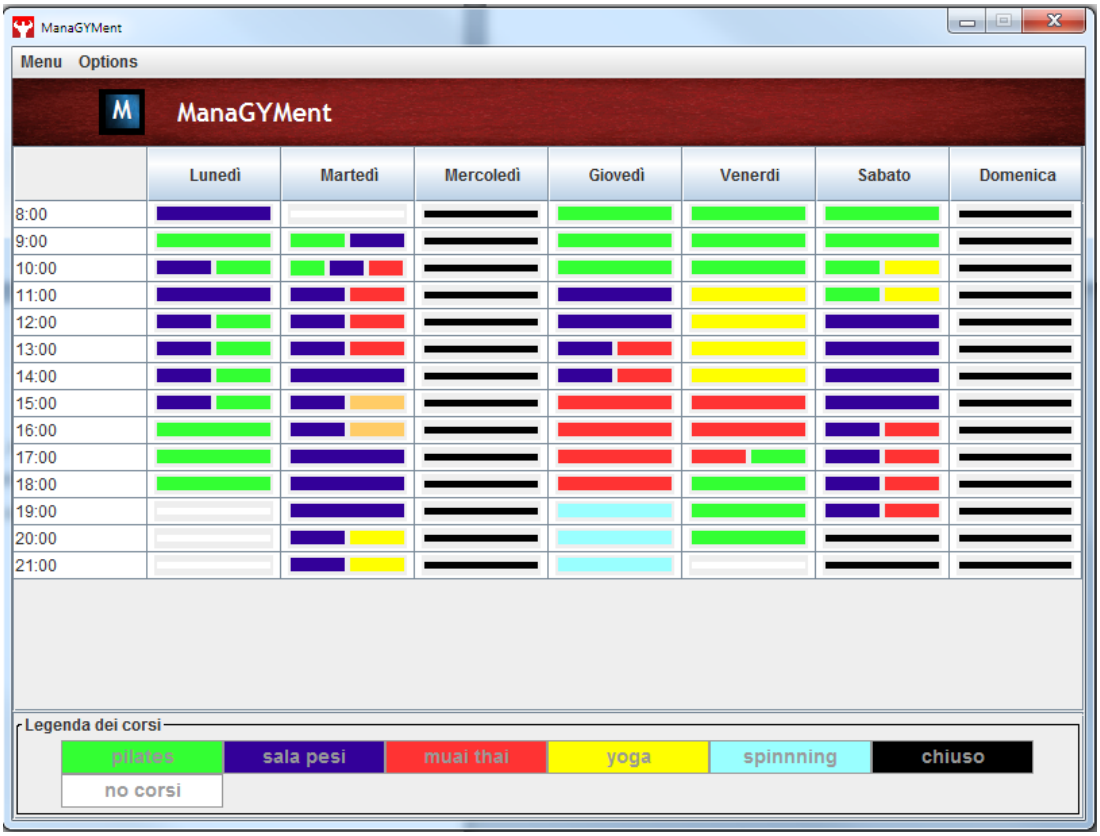
In seguito, inseriti gli impiegati, per assegnare questi ultimi ai corsi tornare sulla pagina della palestra (quella in figura 5.4) e, dopo aver selezionato un corso si aprirà la seguente finestra di dialogo:



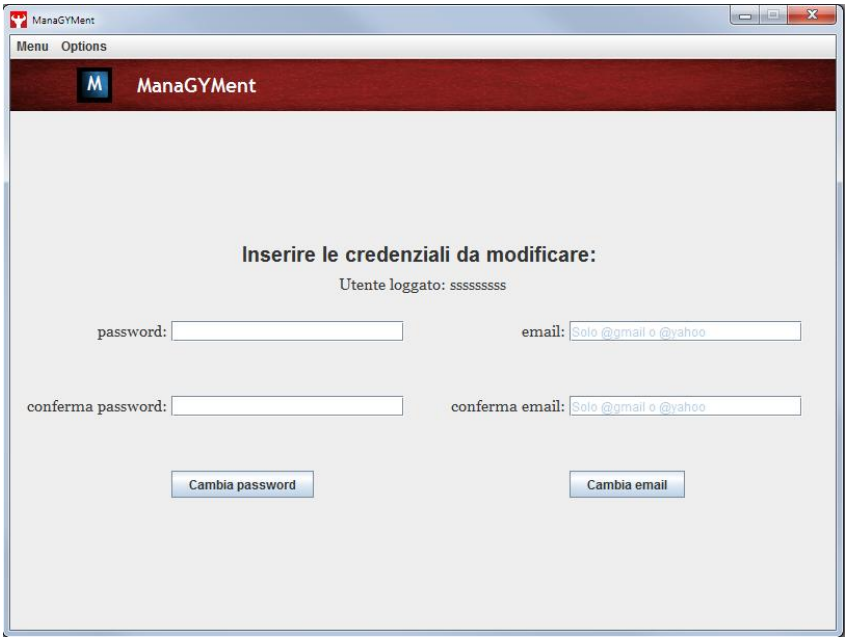
Infine , dopo aver inserito corsi e impiegati rimane da impostare il calendario settimanale della palestra.
Tornare quindi nella pagina Home e premete il tasto sinistro del mouse sul bottone presente nell'intestazione della tabella in base giorno che volete modificare. Si apre una finestra di dialogo come nella seguente figura; da qui è possibile aprire/chiudere la palestra nel determinato giorno, stabilire l'ora di apertura e di chiusura e inserire i corsi con i relativi insegnanti:



Questo è il risultato dopo aver settato alcuni giorni del calendario:



Nel menu principale la sezione PROFILO apre la schermata in figura 5.6, tramite il quale è possibile modificare la password dell'utente e modificare l'account email dalla quale verranno spediti i messaggi.



La sezione email apre la schermata in figura 5.7 addetta all'invio di email a: iscritti, impiegati o entrambi.

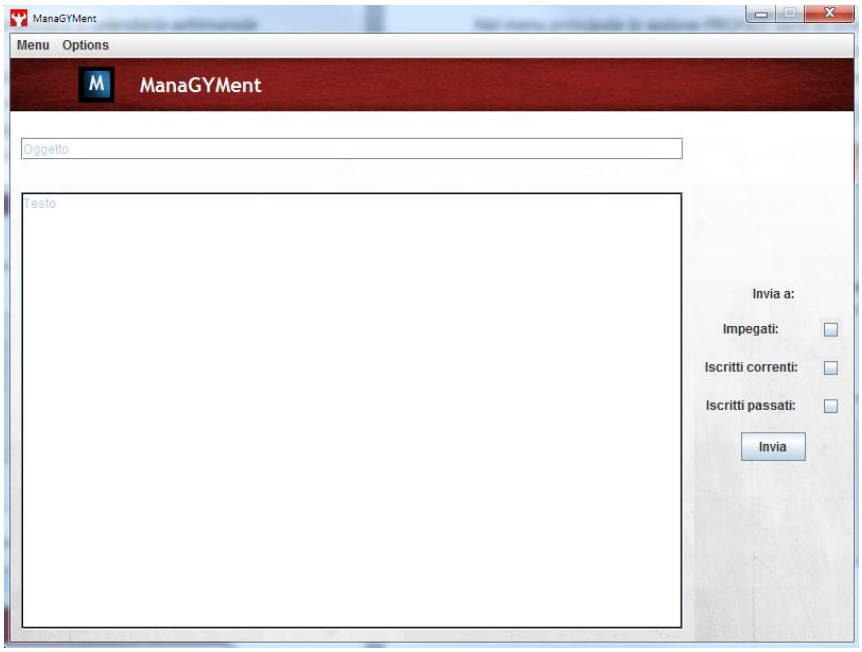


Figura 5.7