Università Politecnica delle Marche – Facoltà di Ingegneria

INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

OOP1617Gruppo01

RELAZIONE

SIMONE ONORI, ELIA PACIONI, RICCARDO SMERILLI, FRANCESCO TALENTO

Sommario

Introduzione

Il progetto è stato sviluppato per rispondere ad una necessità reale del circolo cittadino di Ascoli Piceno: informatizzare la gestione della struttura e dei soci.

Il circolo cittadino si occupa dell'organizzazione di molteplici attività sociali che annualmente propone. Le attività si ispirano al principio di cercare di soddisfare il più possibile gli interessi e le diverse preferenze dei soci. Si spazia quindi dai viaggi, alle gite dedicate al riposo o allo sport sciistico invernale, a mete ancor più localmente caratterizzate, alla ricerca di particolarità e curiosità del territorio. Dagli intrattenimenti danzanti, ai concerti, dalle manifestazioni di carattere culturale, agli incontri conviviali, dalle feste ai giochi come bridge, burraco e biliardo.

Il circolo propone una serie di attività sociali rivolte ai soci e ai loro familiari (coniuge e figli fino ai trentacinque anni di età), inoltre stipula convenzioni con attività commerciali e di ristorazione dando la possibilità di fruire di questi benefit ai soci e a tutti i familiari (compresi i figli di età maggiore ai trentacinque anni)

L'applicazione permette al gestore di inserire soci, gestire l'anagrafica, registrare i pagamenti, le prenotazioni agli eventi e consultarne lo storico.

Il lettore viene invitato a consultare i manuali di:

- installazione e configurazione: necessario per la corretta configurazione della macchina e per l'installazione del programma, semplificando al massimo gli sforzi dell'utente che intende utilizzare il nostro prodotto;
- utente: necessario per utilizzare in modo corretto il programma e per conoscerne tutte le funzioni messe a disposizione.

Analisi del domino d'interesse

Dopo aver intervistato i titolari dell'azienda abbiamo analizzato i processi interni ed abbiamo selezionato gli obiettivi che vorremo raggiungere con il nostro software.

Il nostro obiettivo è progettare un software e una base di dati in grado di gestire il flusso di informazioni interno al circolo.
Dovranno essere gestiti i dati riguardanti:

- i soci con i loro familiari;
- i clienti che non risultano associati;
- lo stato dei pagamenti di tutti i soci;
- lo stato delle prenotazioni delle prenotazioni.

Dovranno essere gestiti anche casi di dimissione o espulsione.

È fondamentale mantenere uno storico dei soci e della loro famiglia anche dopo le dimissioni, poiché i dati costituiscono uno dei patrimoni principali del circolo cittadino.

Soci

I soci iscritti al circolo vengono divisi in quattro tipologie:

- Ordinario
- Straordinario
- Benemerito
- Onorario

I *Soci ordinari* sono tutti i soci che risiedono nella città e hanno meno di cinquanta anni di iscrizione al circolo. Per i residenti nella città ci sono anche delle categorie con alcuni vantaggi e sono:

- *Più giovane*, fino a trentacinque anni di età;
- *Giovane*, dai trentacinque ai quaranta anni di età.

I *soci straordinari* sono i soci che non risiedono nella città e hanno meno di cinquanta anni di iscrizione al circolo.

I *soci benemeriti* sono tutti i soci con più di cinquanta anni di iscrizione al circolo.

I *soci onorari* sono soci a tutti gli effetti, ma non pagano le quote sociali e sono nominati dal consiglio di amministrazione.

Le categorie sociali attribuite sono dinamiche e variano a seconda dei requisiti di ammissione soddisfatti dal socio.

I passaggi di categoria verranno effettuati all'inizio dell'anno seguente al raggiungimento del requisito.

I soci possono decidere di dimettersi ed abbandonare la propria carica sociale, in tal caso il passaggio di stato ad ex-socio avrà effetto immediato. Oltre alla dimissione spontanea possono verificarsi casi di dimissione per cause naturali (decesso) e casi di espulsione.

Un socio che si è dimesso volontariamente può presentare domanda di riammissione in qualsiasi momento.

Quota sociale

Per essere ammesso il socio deve innanzitutto versare una quota associativa variabile.

Tutti i soci pagano una quota mensile variabile in base alla tipologia di appartenenza. Le quote possono subire variazioni anche più volte nel corso dello stesso anno e possono essere versate nelle seguenti modalità:

- Addebito bancario;
- Esattore.

Le quote possono essere pagate con le seguenti soluzioni:

- Unica soluzione ad inizio anno;
- Semestralmente;
- Trimestralmente;
- Mensilmente.

Nel caso in cui dovessero variare le quote di mensilità già versate, la differenza dovrà essere versata alla fine dell'anno;
infatti al termine di ogni anno solare verrà effettuata la chiusura annuale relativa al bilancio di ogni socio, in modo tale da stabilire, tenendo conto delle quote sociale, debiti e crediti di ogni socio.

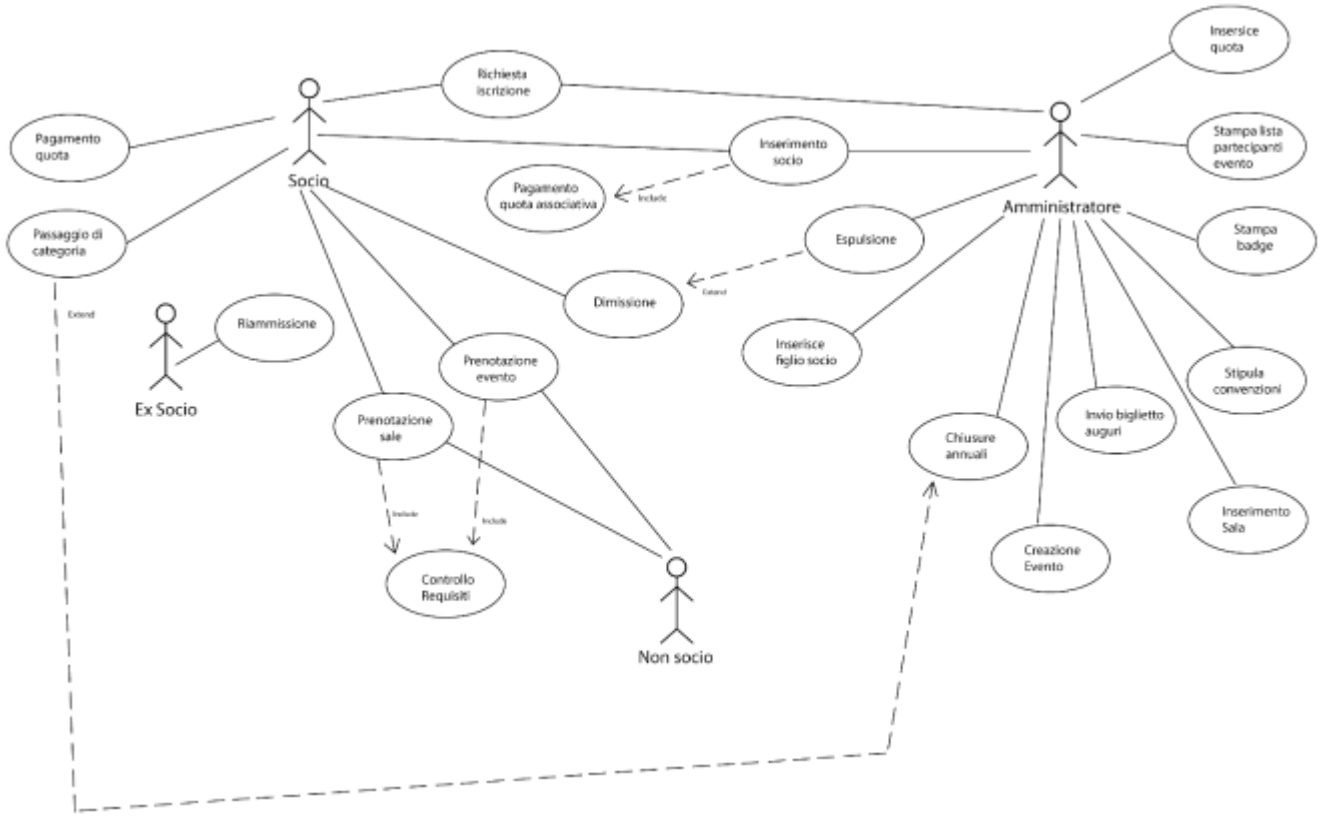
Attività

Le varie attività sociali, organizzate dal circolo, sono rivolte ai soci e ai clienti non associati. Esse prevedono la possibilità di partecipare a feste, viaggi ed eventi, con possibilità di prenotazione delle sale che il circolo mette a disposizione.

Tra gli eventi troviamo *"Le befane"*, quest'ultimo è rivolto esclusivamente ai figli dei soci di età compresa tra i due e i dodici anni, divisi per fasce di età e per sesso, al fine di consegnare i giusti regali.

Strutturazione dei requisiti

Dopo aver studiato il dominio di interesse abbiamo utilizzato i diagrammi UML per strutturare i requisiti.
In particolare il diagramma dei casi d'uso.



Dal diagramma abbiamo individuato i principali attori:

- Amministratore
- Socio
- ExSocio
- NonSocio

L'amministratore è il contabile del circolo cittadino, che si occupa ordinariamente della gestione del circolo. I casi d'uso collegati all'amministratore rappresentano le principali funzionalità che il nostro software dovrà disporre.

Socio è l'attore "più importante" all'interno del nostro progetto, poiché su di lui si basa buona parte del nostro software.

A lui sono associati sette casi d'uso che rappresentano tutte le operazioni svolte da un socio di cui il circolo deve tenere traccia.

L’ex socio dispone di un solo caso d’uso associato: la riammissione; si diventa ex socio dopo le dimissioni da socio.
Il cliente non associato invece può eseguire esclusivamente le prenotazioni.

Abbiamo poi svolto un’analisi orientata ai dati che insieme ai diagrammi UML ci consente di avere una visione globale della nostra realtà d’interesse.

Analisi orientata ai dati

Progettazione del database

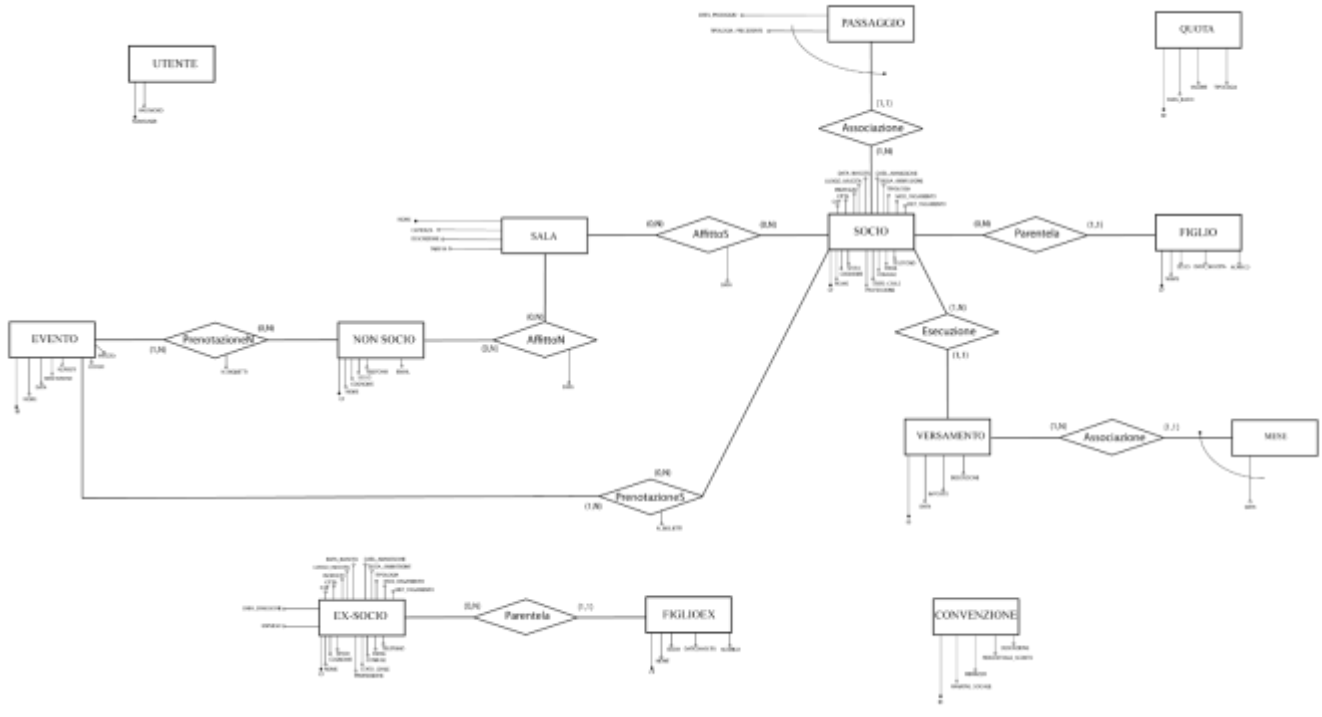
Di fondamentale importanza all'interno della nostra applicazione è la base di dati che si occupa della persistenza dei dati, la cui gestione di questi dati è il motivo principale per il quale abbiamo progettato l'applicazione.

Abbiamo scelto un database di tipo relazione, in particolare il DBMS utilizzato è MySQL. Nella progettazione del database abbiamo deciso di adottare un approccio misto, ibrido tra le strategie top-down e bottom-up in modo tale da sfruttarne i benefici di entrambe le metodologie.

La progettazione è stata svolta nel seguente modo:

- analisi del dominio di interesse e la strutturazione dei requisiti per individuare le entità principali;
- costruzione di un primo scheletro dello schema concettuale per creare le basi per lo schema concettuale effettivo, tramite la strategia top-down;
- attuazione della strategia bottom-up per unire le parti formulate in precedenza;
- realizzazione dello schema concettuale completo, dettagliato, omogeneo e funzionale.

SCHEMA E/R RISTRUTTURATO



L’immagine dello schema è stata allegata al documento nelle dimensioni reali per favorirne la leggibilità.
Per comodità è stato inserito lo schema e/r ristrutturato, ottenuto appunto dalla ristrutturazione dello schema concettuale.

L’entità centrale del nostro database è il *SOCIO*, della quale memorizziamo tutti i dati anagrafici, la professione, i recapiti, i dati relativi alla sua iscrizione e la famiglia. In particolare del coniuge ci interessa memorizzarne solo il nome che è un attributo del socio stesso.
Per quanto riguarda i figli abbiamo l’entità *FIGLIO* della quale memorizziamo i soli dati anagrafici.

Nel momento in cui un socio si dimette (volontariamente o per cause naturali) o viene espulso, tutti i dati, insieme ai dati dei figli, vengono trasferiti nelle relative entità EX-SOCIO e FIGLIOEX.
Abbiamo scelto di traferire i dati relativi agli ex-soci in una tabella separata per ottimizzare i tempi di ricerca dei dati all’interno della tabella *socio*, che è la più utilizzata.

Il socio effettua i versamenti delle quote, il quale, a seconda del metodo di pagamento stabilito, può essere suddiviso in diverse modalità. Un versamento è composto da diversi mesi in base al metodo di pagamento scelto.

Riguardo al versamento memorizziamo la data in cui viene eseguito, l’importo e le note, oltre ai mesi da cui è composto.

Per quanto riguarda la quota, esse varia in relazione alla tipologia del socio e ne memorizziamo la data di inizio, il valore e la tipologia a cui si riferisce. La validità della quota è definita a partire dalla data di inizio, fino alla creazione di una nuova quota per la stessa tipologia. Viene mantenuto uno storico delle quote per gestire al meglio il calcolo delle chiusure annuali, con crediti e debiti dei relativi soci.

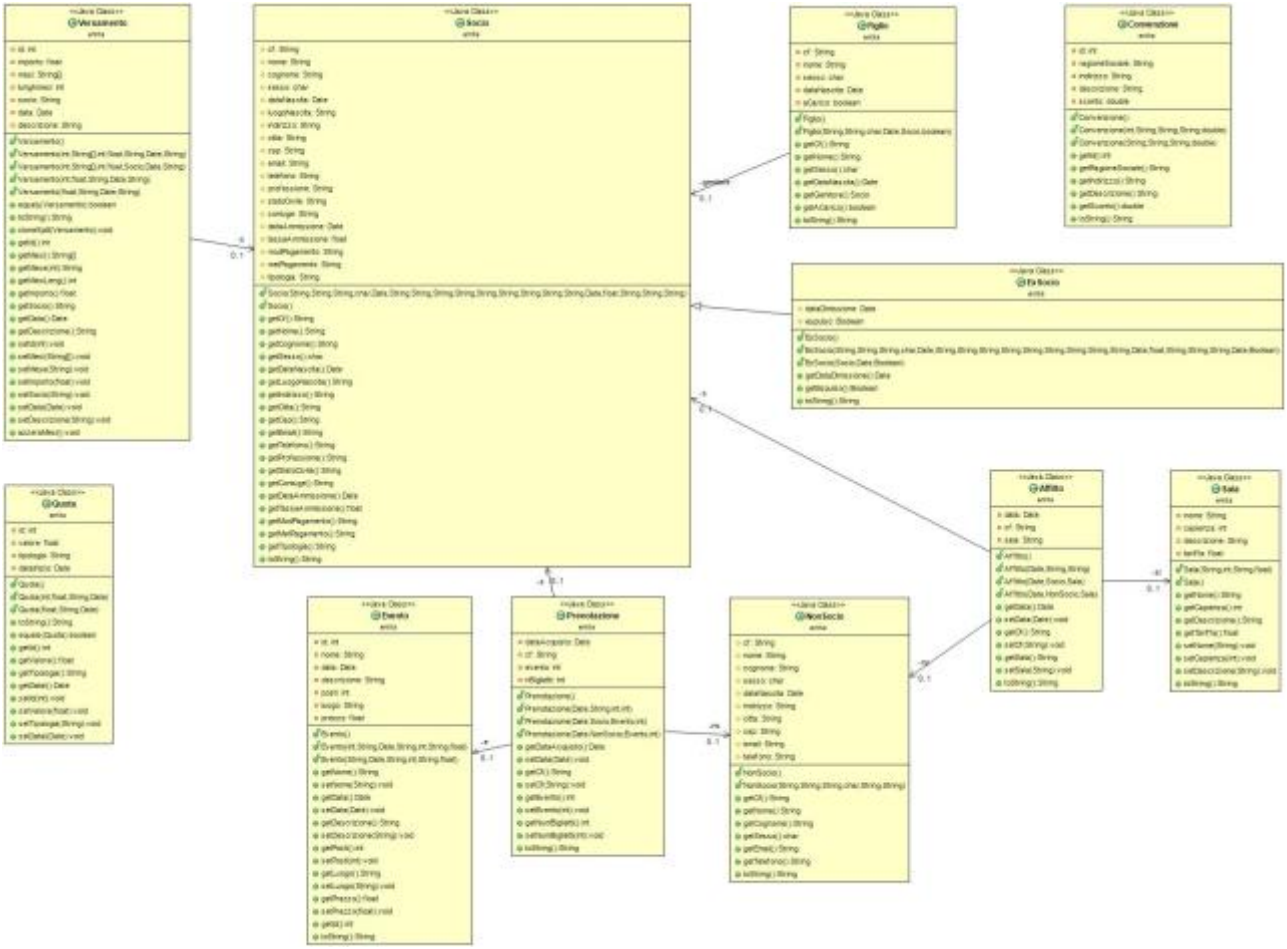
Per quanto riguarda le prenotazioni degli eventi e gli affitti delle sale, possono essere effettuate dai soci e dai clienti non associati (entità NON SOCIO). Per quanto riguarda i clienti non associati memorizziamo i dati indispensabili per poter gestire la prenotazione.

Il circolo stipula anche delle convenzioni con le attività locali e vengono memorizzate nell’apposita entità. Una convenzione è composta dalla ragione sociale e dall’indirizzo dell’azienda con cui è stipulata, la percentuale di sconto applicata e una descrizione aggiuntiva.

Nel database è presente un’entità che non riguarda il dominio di interesse ma è fondamentale per il funzionamento della nostra applicazione, la tabella UTENTE.
Quest’ultima serve per memorizzare gli utenti che possono accedere al software, nella nostra applicazione è stato già inserito di default un utente amministratore poiché l’applicazione verrà utilizzata dal solo contabile del circolo cittadino.

Diagramma delle classi

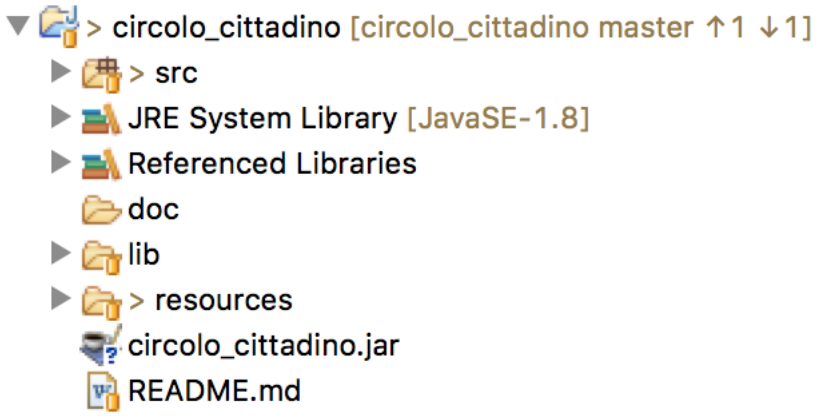
Dal diagramma dei casi d'uso e della progettazione del database si è arrivati alla definizione del diagramma delle classi.



Il diagramma delle classi mette in relazione le classi riguardanti la nostra applicazione, abbiamo scelto di riportare il diagramma esclusivamente delle entità che mappano il database poiché le altre classi sono frutto dell'implementazione del pattern MVC e non dell'analisi della realtà d'interesse.

Struttura del progetto

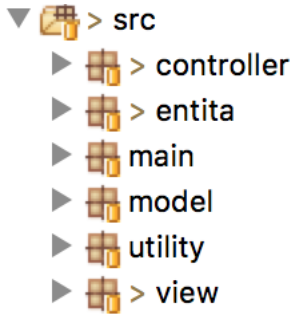
All'interno della cartella principale del progetto ci sono quattro sottocartelle:



- resources contiene:
  - file "circolo.sql" per la creazione del database;
  - file "config.xml" per la configurazione della connessione tra il programma e il database;
  - file "dati.sql" che contiene dei dati di esempio per testare il software e il logo del circolo utilizzato all'interno dell'applicazione;
- lib: contiene le librerie che vengono importate all'interno del programma in fase di configurazione, in particolare la libreria per la connettività al database e le restanti librerie per la manipolazione dei file pdf.
- doc: contiene il *JavaDoc* esportato alla fine del progetto.
- src: è il cuore del progetto e merita una trattazione approfondita.

Struttura della cartella "src"

La cartella "src" rappresenta la parte principale del progetto e contiene tutto il codice sorgente organizzato in package:



Come mostrato nella figura sopra, la cartella src è composta da sei package:

- main: contiene la classe necessaria all'avvio dell'applicazione;
- utility: contiene la classe necessaria per effettuare la connessione al database e la classe usata per la validazione degli input;

- entità: contiene le classi che mappano le entità del database.

I restanti package rappresentano il cuore dell'implementazione del pattern MVC all'interno della nostra applicazione. In particolare i model si occupano di reperire i dati dal database, le view si occupano dell'interfaccia grafica della nostra applicazione, i controller gestiscono gli eventi generati dalle viste.

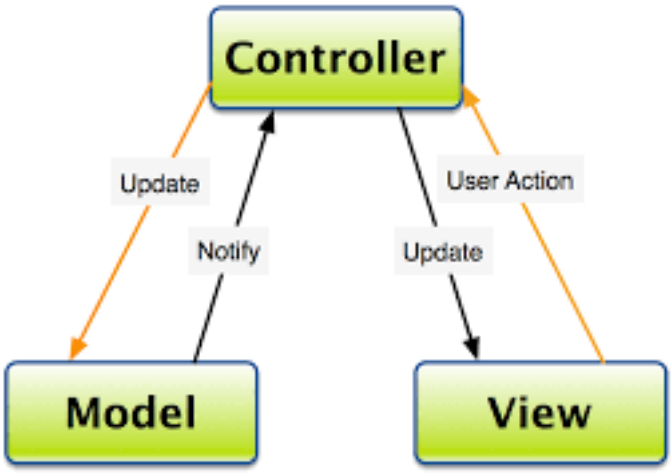
I model e i controller sono stati organizzati per funzionalità.

Architettura e pattern di progetto

Per lo sviluppo dell'applicazione abbiamo deciso di adottare il pattern Model-View-Controller (MVC).

L'MVC è un pattern architetturale il cui obiettivo principale è separare la gestione dei dati dalla logica di presentazione, questa separazione si basa su tre componenti principali:

- **Model (M)**: rappresenta la parte dell'applicazione che si occupa di interagire con il database e di manipolare i dati;
- **View (V)**: rappresenta la GUI della nostra applicazione;
- **Controller (C)**: racchiude la logica applicativa del software, la gestione degli eventi e mette in comunicazione la view e il model



Il pattern MVC è molto diffuso nella programmazione ad oggetti perché, anche se aumenta i tempi necessari allo sviluppo del software, ne semplifica la manutenzione e l'aggiornamento.

Adottando correttamente questo pattern per lo sviluppo del progetto, si aumenta la leggibilità poiché i package e le classi seguono le convenzioni implementative del pattern.

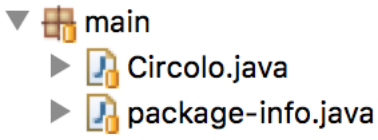
Implementazione

Abbiamo accennato, nel capitolo relativo alla struttura del progetto, all'organizzazione dei package. In questo capitolo andremo più a fondo nell'analisi, guardando all'interno di ogni package le classi che lo compongono.

I file package-info.java non verranno mai tratti approfonditamente poi contengono esclusivamente le descrizioni del package.

Main

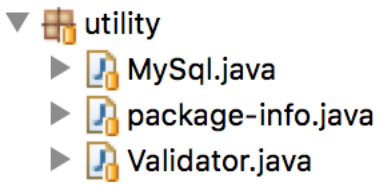
Il package main è quello che si occupa di far partire il programma, contiene la classe circolo che implementa appunto il metodo main.




```
package main;

import java.awt.EventQueue;

import controller.LoginController;

/**
 * @author simoneonori
 * @author eliapacioni
 * @author riccardosmerilli
 * @author francescotalento
 * @version 1.0 Marzo 2017
 *
 * Classe principale del programma, contiene il metodo main(), rappresenta quindi il punto d'ingresso dell'applicazione.
 * Si occupa di chiamare il controller che gestisce il login
 */
public class Circolo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    LoginController controller = new LoginController();
                    controller.controlloLogin();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

}
```

Utility

Il package utility contiene classi utili al funzionamento del nostro programma, in particolare:



MySql

La classe MySql si occupa della comunicazione del nostro programma con il database. Le sue proprietà sono:

```
private String driver;
private String url;
private String utente;
private String password;
private Connection conn = null;
```

Di particolare interesse è il costruttore che si occupa di leggere dal file *config.xml* i parametri di connessione al database.

```
public MySql() {
    try {
        File inputFile = new File("resources/config.xml");
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();

        Document doc = builder.parse(inputFile);
        doc.getDocumentElement().normalize();

        XPath xPath = XPathFactory.newInstance().newXPath();

        String expression = "/database";

        NodeList nodeList = (NodeList) xPath.compile(expression).evaluate(doc, XPathConstants.NODESET);

        Node nNode = nodeList.item(0);
        if (nNode.getNodeType() == Node.ELEMENT_NODE) {
            Element eElement = (Element) nNode;

            driver = eElement.getElementsByTagName("driver").item(0).getTextContent();
            url = eElement.getElementsByTagName("url").item(0).getTextContent();
            utente = eElement.getElementsByTagName("user").item(0).getTextContent();
            password = eElement.getElementsByTagName("password").item(0).getTextContent();
        }
    } catch (ParserConfigurationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (SAXException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (XPathExpressionException e) {
        e.printStackTrace();
    }
}
```

Troviamo poi i metodi: open(), close(), i get e il toString(); rispettivamente si occupano di aprire una connessione con il database, chiudere la connessione con il database, leggere i parametri al di fuori della classe, restituire l'oggetto
in forma testuale.

Validator

La classe Validator non ha proprietà ed è composta esclusivamente da metodi statici che consentono tramite l'utilizzo delle regex di validare la sintassi dell'input.

Essendo i metodi statici non c'è bisogno di istanziare ogni volta la classe, è possibile utilizzare il metodo invocandolo con Validator.nomeMetodo(parametro);

Analizziamo in particolare il seguente metodo:

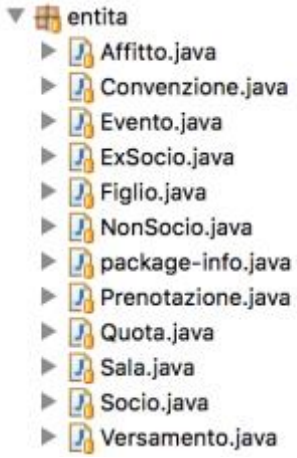
```
public static boolean validaData(String str) {
    Pattern patt = Pattern.compile("(19|20)[0-9]{2}[- /.](0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])");
    Matcher match = patt.matcher(str);
    return match.matches();
}
```

Il metodo nell'immagine si occupa di validare una data in formato AAAA/MM/GG, riceve in ingresso una stringa e restituisce un valore booleano in base all'esito della validazione.
Il metodo in questione utilizza le regex, in particolare definisce un preciso pattern e controlla se quest'ultimo è rispettato dalla stringa da validare.

I metodi di questa classe utilizzano tutti la stessa filosofia, per questo ci limiteremo ad approfondire solo questo metodo.

Entità

Il package entità contiene tutte le classi necessarie a mappare le entità presenti nel database:



Tutte le classi del package entità non contengono metodi al di fuori dei cotruttori, dei get/set, toString() ed equals(), sono classi che servono esclusivamente a memorizzare i dati estratti dal database. I metodi che utilizzano i dati di queste classi sono presenti nel model e nel controller come previsto dal pattern MVC.

Analizziamo solamente la classe Socio perché tutte le classi seguono la stessa logica implementativa, sono quindi strutturate allo stesso modo:

```
protected String cf;
protected String nome;
protected String cognome;
protected char sesso;
protected Date dataNascita;
protected String luogoNascita;
protected String indirizzo;
protected String citta;
protected String cap;
protected String email;
protected String telefono;
protected String professione;
protected String statoCivile;
protected String coniuge;
protected Date dataAmmissione;
protected float tassaAmmissione;
protected String modPagamento;
protected String metPagamento;
protected String tipologia;
```

Le proprietà del socio rispecchiano gli attributi dell'entità socio presente nel database.

Sono presenti il costruttore senza parametri ed il costruttore parametrico del socio, hanno il compito di inizializzare le proprietà rispettivamente con valori di default il primo e con i valori passati come parametri il secondo.

```java
 * Costruttore parametrico del socio, inizializza tutte le proprietà con i valori passati
public Socio(String codice, String name, String surname, char sex, Date dateB, String placeB, String address,
        String city, String postalCode, String mail, String tel, String profession, String civilStatus,
        String spouse, Date dateAmmission, float taxAmmission, String modPay, String metPay, String type) {
    cf = codice;
    nome = name;
    cognome = surname;
    sesso = sex;
    dataNascita = dateB;
    luogoNascita = placeB;
    indirizzo = address;
    citta = city;
    cap = postalCode;
    email = mail;
    telefono = tel;
    professione = profession;
    statoCivile = civilStatus;
    coniuge = spouse;
    dataAmmissione = dateAmmission;
    tassaAmmissione = taxAmmission;
    modPagamento = modPay;
    metPagamento = metPay;
    tipologia = type;
}

 * Costruttore del socio senza parametri
public Socio() {
```

Abbiamo poi i metodi get che ci consentono di leggere i valori delle proprietà dall'esterno della classe:

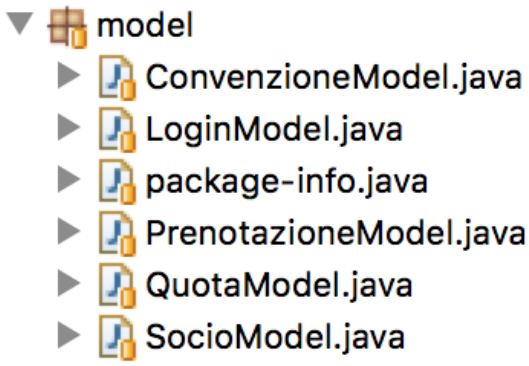
```java
 * Costruttore del socio senza parametri
public Socio() {

 * @return codice fiscale del socio (String)
public String getCf() {

 * @return nome del socio (String)
public String getNome() {

 * @return cognome del socio (String)
public String getCognome() {

 * @return sesso del socio (char)
public char getSesso() {

 * @return data di nascita del socio (Date)
public Date getDataNascita() {

 * @return luogo di nascita del socio (String)
public String getLuogoNascita() {

 * @return indirizzo del socio (String)
public String getIndirizzo() {

 * @return città di residenza del socio (String)
public String getCitta() {

 * @return cap del socio (String)
public String getCap() {

 * @return indirizzo email del socio (String)
public String getEmail() {

 * @return numero di telefono del socio (String)
public String getTelefono() {

 * @return professione del socio (String)
public String getProfessione() {

 * @return statoCivile (String)
public String getStatoCivile() {

 * @return nome del coniuge (String)
public String getConiuge() {

 * @return data ammissione al circolo (Date)
public Date getDataAmmissione() {

 * @return tassa ammissione al circolo (float)
public float getTassaAmmissione() {

 * @return modalità di pagamento scelta dal socio (String)
public String getModPagamento() {

 * @return metodo di pagamento scelto dal socio (String)
public String getMetPagamento() {

 * @return tipologia del socio (String)
public String getTipologia() {
```

Ed infine il metodo toString() che ci restituisce una rappresentazione testuale dell'oggetto in forma di stringa: è molto utile ad esempio per le stampe.

```java
 * Override del metodo toString()
public String toString() {
```

Model

Il package model si occupa di manipolare i dati, in particolare di estrarli, inserirli e cancellarli dal database. I model sono divisi per macro-funzionalità.



ConvenzioniModel: si occupa di gestire l'inserimento, la modifica e la cancellazione e la ricerca delle convenzioni stipulate dal circolo cittadino.

LoginModel: si occupa di controllare se l'utente può essere abilitato, confrontando i dati inviati dal controller con quelli presenti nel database.

PrenotazioneModel: gestisce il reperimento, l'inserimento, la modifica e la cancellazione dei dati che riguardano le prenotazioni dei biglietti per gli eventi, gli affitti delle sale

QuotaModel: si occupa di effettuare le operazioni, sulle quote e i versamenti che riguardano i dati e il database

SocioModel: gestisce l'inserimento, la modifica, la cancellazione e il reperimento delle informazioni riguardanti soci, non soci, ex soci e familiari.

Trattiamo in maniera approfondita il SocioModel:

```
MySql db = null;
```

L'unica proprietà di questa classe è l'oggetto db di tipo MySql che ci permetterà di aprire, gestire e chiudere la connessione con il database

```
/**
 * Costruttore del SocioModel inizializza l'oggetto MySql
 */
public SocioModel() {
    db = new MySql();
}
```

Il costruttore si occupa di istanziare l'oggetto db.

Inserimento

```java
 * Metodo che si occupa dell'inserimento di un nuovo socio nel database
public boolean inserisciSocio(Socio n) {
    db.open();
    PreparedStatement st = null;
    boolean esito = false;
    String query = "INSERT INTO socio(cf, nome, cognome, sesso, data_nascita, luogo_nascita, indirizzo, citta, cap, email, telefono, "
            + "professione, stato_civile, coniuge, data_ammissione, tassa_ammissione, mod_pagamento, met_pagamento, tipologia)"
            + " VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)";
    try {
        st = db.getConn().prepareStatement(query);
        st.setString(1, n.getCf());
        st.setString(2, n.getNome());
        st.setString(3, n.getCognome());
        st.setString(4, Character.toString(n.getSesso()));
        st.setDate(5, n.getDataNascita());
        st.setString(6, n.getLuogoNascita());
        st.setString(7, n.getIndirizzo());
        st.setString(8, n.getCitta());
        st.setString(9, n.getCap());
        st.setString(10, n.getEmail());
        st.setString(11, n.getTelefono());
        st.setString(12, n.getProfessione());
        st.setString(13, n.getStatoCivile());
        st.setString(14, n.getConiuge());
        st.setDate(15, n.getDataAmmissione());
        st.setFloat(16, n.getTassaAmmissione());
        st.setString(17, n.getModPagamento());
        st.setString(18, n.getMetPagamento());
        st.setString(19, n.getTipologia());

        if (st.executeUpdate() == 1)
            esito = true;
        st.close();
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } finally {
        db.close();
    }
    return esito;
}
```

Il metodo sopraesposto si occupa di effettuare l'inserimento di un nuovo socio nel database, prende come parametro un socio e tramite una query parametrizzata effettua l'inserimento.
Per gestire le eccezioni la query viene effettuata all'interno di un try-catch, nel finally invece viene sempre effettuata la chiusura della connessione, a prescindere dall'esito del blocco try-catch.
Il metodo è di tipo booleano e ritorna il valore in base all'esito dell'inserimento.

Eliminazione

```java
 * Metodo che si occupa dell'eliminazione di un socio dal database
public boolean eliminaSocio(Socio n) {
    boolean esito = false;
    db.open();
    PreparedStatement st = null;
    String query = "DELETE FROM socio WHERE cf = ?";
    try {
        st = db.getConn().prepareStatement(query);
        st.setString(1, n.getCf());

        if (st.executeUpdate() == 1)
            esito = true;
        st.close();

    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } finally {
        db.close();
    }

    return esito;
}
```

Questo metodo si occupa dell'eliminazione di un socio dal database. Accetta come parametro un socio e tramite una query parametrizzata effettua l'eliminazione.

Le operazioni di eliminazione sono previste solo in caso di errore nell'inserimento, tutti i dati all'interno del database fanno parte del patrimonio del circolo, pertanto non verranno mai eliminati.

Se un socio si dimette o viene espulso, i suoi dati verranno memorizzati nell'apposita tabella ex-socio.

Modifica

```java
 * Metodo che si occupa della modifica di un socio nel database
public boolean modificaSocio(Socio n, String cf) {
    boolean esito = false;
    db.open();
    PreparedStatement st = null;
    String query = "UPDATE socio SET cf = ?, nome = ?, cognome = ?, sesso = ?, data_nascita = ?, luogo_nascita = ?, indirizzo = ?, citta = ?, cap = ?, email = ?, telefono = ?, "
            + "professione = ?, stato_civile = ?, coniuge = ?, data_ammissione = ?, tassa_ammissione = ?, mod_pagamento = ?, met_pagamento = ?, tipologia = ? WHERE cf = ?;";
    try {
        st = db.getConn().prepareStatement(query);
        st.setString(1, n.getCf());
        st.setString(2, n.getNome());
        st.setString(3, n.getCognome());
        st.setString(4, Character.toString(n.getSesso()));
        st.setDate(5, n.getDataNascita());
        st.setString(6, n.getLuogoNascita());
        st.setString(7, n.getIndirizzo());
        st.setString(8, n.getCitta());
        st.setString(9, n.getCap());
        st.setString(10, n.getEmail());
        st.setString(11, n.getTelefono());
        st.setString(12, n.getProfessione());
        st.setString(13, n.getStatoCivile());
        st.setString(14, n.getConiuge());
        st.setDate(15, n.getDataAmmissione());
        st.setFloat(16, n.getTassaAmmissione());
        st.setString(17, n.getModPagamento());
        st.setString(18, n.getMetPagamento());
        st.setString(19, n.getTipologia());
        st.setString(20, cf);

        if (st.executeUpdate() == 1)
            esito = true;
        st.close();
    } catch (SQLException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } finally {
        db.close();
    }

    return esito;
}
```

Il metodo per la modifica è simile a quello dell'inserimento, la differenza sostanziale sta nel fatto che la query è di update e non di inserimento. Per il resto viene gestito nello stesso modo.

Accetta in ingresso due parametri, il socio da aggiornare e il codice fiscale prima dell'aggiornamento. In questo modo se si commette un errore nel codice fiscale è possibile aggiornarlo anche se è chiave primaria.

Elenco Soci

```
 * Metodo che si occupa di recuperare l'elenco di tutti i soci nel database
public ArrayList<Socio> elencoSoci() {
    ArrayList<Socio> soci = new ArrayList<Socio>();
    Statement st;
    try {
        db.open();
        st = db.getConn().createStatement();
        String query = "SELECT * FROM socio;";
        ResultSet res = st.executeQuery(query);
        while (res.next()) {
            soci.add(new Socio(res.getString("cf"), res.getString("nome"), res.getString("cognome"),
                    res.getString("sesso").charAt(0), res.getDate("data_nascita"), res.getString("luogo_nascita"),
                    res.getString("indirizzo"), res.getString("citta"), res.getString("cap"),
                    res.getString("email"), res.getString("telefono"), res.getString("professione"),
                    res.getString("stato_civile"), res.getString("coniuge"), res.getDate("data_ammissione"),
                    res.getFloat("tassa_ammissione"), res.getString("mod_pagamento"),
                    res.getString("met_pagamento"), res.getString("tipologia")));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return soci;
}
```

Il metodo in figura permette di recuperare tutti i soci dal database e li inserisce in un ArrayList<Socio> che viene ritornato dal metodo.

Ricerca Socio

```
 * Metodo che si occupa di estrarre un determinato socio dal database
private Socio cercaSocio(String cf) {
    Socio socio = null;
    PreparedStatement st;
    String query = "SELECT * FROM socio WHERE cf = ?";
    try {
        db.open();
        st = db.getConn().prepareStatement(query);
        st.setString(1, cf);
        ResultSet res = st.executeQuery();
        if (res.next()) {
            socio = new Socio(res.getString("cf"), res.getString("nome"), res.getString("cognome"),
                    res.getString("sesso").charAt(0), res.getDate("data_nascita"), res.getString("luogo_nascita"),
                    res.getString("indirizzo"), res.getString("citta"), res.getString("cap"),
                    res.getString("email"), res.getString("telefono"), res.getString("professione"),
                    res.getString("stato_civile"), res.getString("coniuge"), res.getDate("data_ammissione"),
                    res.getFloat("tassa_ammissione"), res.getString("mod_pagamento"),
                    res.getString("met_pagamento"), res.getString("tipologia"));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        db.close();
    }

    return socio;
}
```

Questo metodo si occupa di cercare un socio a partire dal suo codice fiscale, il valore di ritorno è un oggetto socio.

Passaggio di categoria

Il passaggio di categoria avviene per i soci che soddisfano le seguenti condizioni condizioni

Tipologia di partenza	Tipologia Futura	Criterio
Ordinario	Benemerito	50 anni di iscrizione al circolo
Giovane	Ordinario	Più di 40 anni d'età
Più Giovane	Giovane	Tra i 35 e i 40 anni d'età

Il seguente metodo si occupa quindi di determinare quali soci soddisfano i criteri per effettuare il passaggio di categoria, li inserisce in un ArrayList<Socio> che sarà il valore di ritorno del metodo.

```
 * Metodo che si occupa di recuperare tutti i soci che devono effettuare il passaggio di categoria secondo i criteri stabiliti
public ArrayList<Socio> passaggioCategoria() {
    ArrayList<Socio> soci = new ArrayList<Socio>();
    Statement st;
    try {
        db.open();
        st = db.getConn().createStatement();
        String query = "SELECT * FROM socio WHERE (DATEDIFF(NOW(), data_ammissione)>=18250 AND tipologia = 'ORDINARIO') "
                + "OR (DATEDIFF(NOW(), data_nascita)>=14600 AND tipologia = 'GIOVANE')"
                + "OR (DATEDIFF(NOW(), data_nascita)>=12775 AND tipologia = 'PIU GIOVANE');";
        ResultSet res = st.executeQuery(query);
        while (res.next()) {
            soci.add(new Socio(res.getString("cf"), res.getString("nome"), res.getString("cognome"),
                    res.getString("sesso").charAt(0), res.getDate("data_nascita"), res.getString("luogo_nascita"),
                    res.getString("indirizzo"), res.getString("citta"), res.getString("cap"),
                    res.getString("email"), res.getString("telefono"), res.getString("professione"),
                    res.getString("stato_civile"), res.getString("coniuge"), res.getDate("data_ammissione"),
                    res.getFloat("tassa_ammissione"), res.getString("mod_pagamento"),
                    res.getString("met_pagamento"), res.getString("tipologia")));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        db.close();
    }

    return soci;
}
```

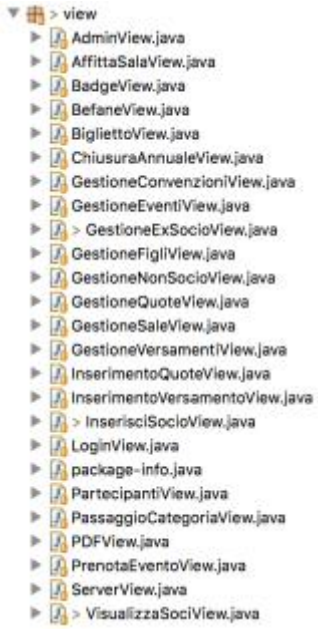
Subito dopo questo metodo vengono effettivamente attivati i passaggi di categoria per ogni singolo socio, tenendo traccia del passaggio nell'apposita tabella del database *passaggio*:

```
 * Metodo che si occupa di rendere effettivi i passaggi di categoria dei soci, modifica la tipologia del socio nel database e tiene traccia del passaggio inserendo i dati relativi ne
public boolean effettuaPassaggioCategoria(Socio n) {
    boolean esito = false;
    PreparedStatement st;
    String tip = null;
    switch (n.getTipologia()) {
    case ("ORDINARIO"):
        tip = "BENEMERITO";
        break;
    case ("GIOVANE"):
        tip = "ORDINARIO";
        break;
    case ("PIU GIOVANE"):
        tip = "GIOVANE";
        break;
    }
    try {
        db.open();
        String query = "UPDATE socio SET tipologia = ? WHERE cf = ? ";
        st = db.getConn().prepareStatement(query);
        st.setString(1, tip);
        st.setString(2, n.getCf());
        st.executeUpdate();
        query = "INSERT INTO passaggio (data_passaggio, tipologia_precedente, socio) VALUES(?,?,?);";
        st = db.getConn().prepareStatement(query);
        st.setDate(1, Date.valueOf(LocalDate.now()));
        st.setString(2, n.getTipologia());
        st.setString(3, n.getCf());
        st.executeUpdate();
        esito = true;
        st.close();
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        db.close();
    }

    return esito;
}
```

Abbiamo deciso di tenere traccia dei vari passaggi di categoria per poter facilitare il calcolo delle chiusure e mantenere uno storico

View

Il package view si occupa della gestione della GUI, le view sono basate su Swing, al suo interno troviamo tutte le classi necessarie per creare le interfacce grafiche necessarie:



In particolare analizziamo la *VisualizzaSociView*

Le proprietà comprendono tutti i componenti necessari per visualizzare tutte le informazioni del socio e per inoltrare gli eventi desiderati.

```
private JFrame frame;
private JList<Socio> list;
private JTextField cf;
private JTextField nome;
private JTextField cognome;
private JTextField dataNascita;
private JTextField luogoNascita;
private JRadioButton rdbtnUomo;
private JRadioButton rdbtnDonna;
private ButtonGroup sesso;
private JTextField indirizzo;
private JTextField citta;
private JTextField cap;
private JTextField email;
private JTextField telefono;
private JTextField professione;
private JComboBox<String> statoCivile;
private JTextField coniuge;
private JTextField dataAmmissione;
private JTextField tassaAmmissione;
private JComboBox<String> modPagamento;
private JComboBox<String> metPagamento;
private JComboBox<String> tipologia;
private JButton btnDashboard;
private JButton btnModifica;
private JButton btnDiventaExsocio;
private JButton btnEspelli;
private JButton btnElimina;
private DefaultListModel<Socio> dlm;
private ScrollPane scrollPane;
private JButton btnAggiorna;
private JButton btnAnnulla;
```

Il costruttore si occupa di istanziare tutti componenti definiti come proprietà, per motivi di spazio ne viene riportata solamente una parte:

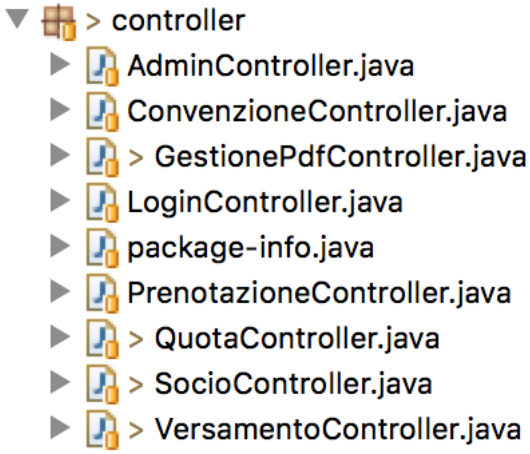
```
 * @param soci, elenco dei soci
public VisualizzaSociView(ArrayList<Socio> soci) {
    frame = new JFrame("Elenco soci del Circolo Cittadino di Ascoli Piceno");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setBounds(100, 100, 800, 600);
    frame.setResizable(false);
    frame.getContentPane().setLayout(null);

    JLabel lblElencoSoci = new JLabel("Elenco Soci ");
    lblElencoSoci.setBounds(341, 20, 75, 16);
    frame.getContentPane().add(lblElencoSoci);

    list = new JList<Socio>();
    dlm = new DefaultListModel<Socio>();
    soci.stream().forEach((s)->{
        dlm.addElement(s);
    });
    list.setModel(dlm);

    scrollPane = new ScrollPane();
    scrollPane.setBounds(6, 54, 228, 518);
    frame.getContentPane().add(scrollPane);
    scrollPane.add(list);

    JLabel lblCodiceFiscale = new JLabel("Codice Fiscale");
    lblCodiceFiscale.setBounds(253, 95, 97, 16);
    frame.getContentPane().add(lblCodiceFiscale);

    cf = new JTextField();
    cf.setBounds(362, 90, 138, 26);
    cf.setColumns(10);
    cf.setEnabled(false);
    frame.getContentPane().add(cf);
```

Ci sono poi i metodi get per accedere dall'esterno ai componenti della view, anche in questo caso ne riportiamo solo alcuni:

```
 * @return btnDiventaExsocio bottone per dimettere il socio (JButton)
public JButton getBtnDiventaExsocio() {

 * @return btnEspelli (JButton) bottone per l'espulsione del socio
public JButton getBtnEspelli() {

 * @return btnElimina (JButton) bottone per l'eliminazione del socio
public JButton getBtnElimina() {

 * @return btnAnnulla (JButton) bottone per annullare le modifiche
public JButton getBtnAnnulla() {

 * @return scrollPane (ScrollPane) visualizza la lista
public ScrollPane getScrollPane() {
```

Controller

Passiamo ora ad analizzare il controller che è un po' l'arbitro della situazione, si occupa di gestire gli eventi scatenati dalla view, richiedere i dati al model e aggiornare di conseguenza la view:



Analizziamo il SocioController che si occupa di gestire gli eventi riguardanti: Socio, NonSocio, ExSocio, Figlio.
Esso il controller molto grande, per comodità analizziamo solo alcuni metodi.

```
private SocioModel model;

 * Costruttore senza parametri del SocioController, istanzia il SocioModel
public SocioController() {
    model = new SocioModel();
}
```

L'unica proprietà del SocioController è il model di tipo SocioModel, ci servirà per poter chiamare i metodi relativi al model, esso viene istanziato nel costruttore della classe poiché è utilizzato da tutti i suoi metodi.

```
 * Metodo che gestisce gli eventi della InserisciSocioView,
public void inserimentoSocio() {
    InserisciSocioView view = new InserisciSocioView();
    view.getFrame().setVisible(true);
    view.getStatoCivile().addItemListener(new ItemListener() {

    view.getBtnInserisci().addMouseListener(new MouseAdapter() {

    view.getBtnDashboard().addMouseListener(new MouseAdapter() {
}
```

Ogni metodo gestisce gli eventi relativi ad una view, nella foto sopra ad esempio inserimento() gestisce gli eventi della InserisciSocioview.
Per la gestione degli eventi vengono usati dei Listener che sono in sostanza degli ascoltatori, pronti ad attivarsi al verificarsi dell'evento specificato.
Analizziamo in particolare il listener sul botton di inserimento

```
view.getBtnInserisci().addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        char sex;
        if (view.getRdbtnUomo().isSelected())
            sex = 'M';
        else
            sex = 'F';

        String cf = view.getCf().getText().toUpperCase();
        String nome = view.getNome().getText().toUpperCase();
        String cognome = view.getCognome().getText().toUpperCase();
        String dataNascita = view.getDataNascita().getText();
        String luogoNascita = view.getLuogoNascita().getText().toUpperCase();
        String indirizzo = view.getIndirizzo().getText().toUpperCase();
        String citta = view.getCitta().getText().toUpperCase();
        String cap = view.getCap().getText();
        String email = view.getEmail().getText().toUpperCase();
        String telefono = view.getTelefono().getText();
        String professione = view.getProfessione().getText().toUpperCase();
        String statoCivile = view.getStatoCivile().getSelectedItem().toString().toUpperCase();
        String coniuge = null;
        if (view.getConiuge().getText().length() > 0) {
            coniuge = view.getConiuge().getText().toUpperCase();
        }
        String dataAmmissione = view.getDataAmmissione().getText();
        String tassaAmmissione = view.getTassaAmmissione().getText();
        String modPagamento = view.getModPagamento().getSelectedItem().toString().toUpperCase();
        String metPagamento = view.getMetPagamento().getSelectedItem().toString().toUpperCase();
        String tipologia = view.getTipologia().getSelectedItem().toString().toUpperCase();
```

Quando l'evento viene scatenato, viene chiamato il metodo mouseClicked(Mouse Event e) che è un override del metodo fornito di base.
All'interno di questo metodo vengo prima di tutto recuperati i valori dalla vista, tramite gli appositi get.

Dopo aver recuperato i dati verranno validati:

```
        boolean validazione = true;
        if (!Validator.validaCf(cf)) {
            view.getCf().setBackground(Color.red);
            validazione = false;
        } else {
            if (view.getCf().getBackground() == Color.red)
                view.getCf().setBackground(Color.white);
        }
        if (!Validator.validaAnagrafica(nome)) {
            view.getNome().setBackground(Color.red);
            validazione = false;
        } else {
            if (view.getNome().getBackground() == Color.red)
                view.getNome().setBackground(Color.white);
        }
        if (!Validator.validaAnagrafica(cognome)) {
            view.getCognome().setBackground(Color.red);
            validazione = false;
        } else {
            if (view.getCognome().getBackground() == Color.red)
                view.getCognome().setBackground(Color.white);
        }
        if (!Validator.validaData(dataNascita)) {
            view.getDataNascita().setBackground(Color.red);
            validazione = false;
        } else {
            if (view.getDataNascita().getBackground() == Color.red)
                view.getDataNascita().setBackground(Color.white);
        }
        if (!Validator.validaAnagrafica(luogoNascita) || luogoNascita.length() > 35) {
            view.getLuogoNascita().setBackground(Color.red);
            validazione = false;
        } else {
            if (view.getLuogoNascita().getBackground() == Color.red)
                view.getLuogoNascita().setBackground(Color.white);
        }
        if (!Validator.validaIndirizzo(indirizzo)) {
            view.getIndirizzo().setBackground(Color.red);
            validazione = false;
        } else {
            if (view.getIndirizzo().getBackground() == Color.red)
                view.getIndirizzo().setBackground(Color.white);
        }
```

solo se la validazione andrà a buon fine, verrà chiamato il rispettivo metodo del model che si occupa dell'inserimento di un socio nel database.
A quest'ultimo verrà passato come parametro proprio un oggetto di tipo socio, creato in modo anonimo proprio al momento della chiamata del metodo.

```
        if (validazione) {
            boolean esito = model.inserisciSocio(new Socio(cf, nome, cognome, sex, Date.valueOf(dataNascita),
                    luogoNascita, indirizzo, citta, cap, email, telefono, professione, statoCivile, coniuge,
                    Date.valueOf(dataAmmissione), Float.valueOf(tassaAmmissione), modPagamento, metPagamento,
                    tipologia));

            if (esito) {
                JOptionPane.showMessageDialog(view.getFrame().getContentPane(), "Inserimento Effettuato");
                inserimentoSocio();
                view.getFrame().dispose();
            } else {
                JOptionPane.showMessageDialog(view.getFrame().getContentPane(), "Inserimento Non Effettuato");
            }

        } else {
            JOptionPane.showMessageDialog(view.getFrame().getContentPane(),
                    "Campi non validi, modificare i campi contrassegnati in rosso");
        }
    }
});
```

Verrà poi visualizzato un messaggio relativo all'esito dell'inserimento, quest'ultimo è verificabile tramite il valore di ritorno del metodo.

In caso non sia andata a buon fine la validazione e, non è stato quindi chiamato il model, verrà visualizzato un messaggio di errore chiedendo di modificare i campi contrassegnati dal colore rosso.

Oltre al Listener per il bottone dell'inserimento ce n'è uno che è comune a quasi tutte le view del nostro programma: il listener per il ritorno alla dashboard

```
view.getBtnDashboard().addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        AdminController adminController = new AdminController();
        adminController.controlloEvento();
        view.getFrame().dispose();
    }
});
```

Al click del bottone dashboard il metodo mouseClicked si occupa di istanziare l'adminController e di chiamare il rispettivo metodo per il controllo degli eventi. Un volta visualizzata la view dell'admin verrà poi chiuso il frame precedente che in questo caso riguarda la gestione del socio.

Strumenti di programmazione utilizzati

La programmazione orientata agli oggetti (Object Oriented Programming) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi.

Il paradigma ad oggetti ha i seguenti punti cardine:

- Incapsulamento;
- Ereditarietà;
- Polimorfismo.

Incapsulamento

L'incapsulamento è la proprietà per cui i dati che definiscono lo stato interno di un oggetto e i metodi che ne definiscono la logica sono accessibili ai metodi dell'oggetto stesso, mentre non sono visibili ai client. Per alterare lo stato interno dell'oggetto, è necessario invocarne i metodi pubblici, ed è questo lo scopo principale dell'incapsulamento. Infatti, se gestito opportunamente, esso permette di vedere l'oggetto come una black-box, cioè una "scatola nera" di cui, attraverso l'interfaccia, è noto cosa fa, ma non come lo fa.

Ereditarietà

L'ereditarietà è fondamentale nella programmazione ad oggetti poiché consente la creazione di classificazioni gerarchiche.
Attraverso l'ereditarietà è possibile creare una classe generale che definisca tratti comuni a una serie di elementi correlati. Questa classe può essere ereditata da altre classi più specifiche, ciascuna delle quali aggiunge altri elementi.
In Java una classe ereditata di chiama superclasse, la classe che eredita sottoclasse.

Nel nostro progetto l'ereditarietà è stata usata tra socio ed ex-socio, quest'ultimo estende il socio aggiungendo alcune proprietà:

```java
package entita;

import java.sql.Date;

 * @author simoneponsi
public class ExSocio extends Socio {
    protected Date dataDimissione;
    protected Boolean espulso;

    /**
     * Costruttore dell'exsocio senza parametri
     *
     */
    public ExSocio() {
        super();
        dataDimissione = null;
        espulso = null;
    }

     * Costruttore parametrico dell'exsocio, inizializza tutte le proprietà con i valori passati
    public ExSocio(String codice, String name, String surname, char sex, Date date8, String place8, String address,

     * @param n socio che deve diventare exsocio
    public ExSocio(Socio n, Date dimissionDate, Boolean expelled){

     * @return data di dimissione dal circolo (Date)
    public Date getDataDimissione() {

     * @return valore booleano se il socio è stato o no espulso
    public Boolean getEspulso() {

     * Override del metodo toString()
    public String toString() {
}
```

Polimorfismo

Il polimorfismo è particolarmente utile quando la versione del metodo da eseguire viene scelta sulla base del tipo di oggetto effettivamente contenuto in una variabile a runtime (invece che al momento della compilazione). Questa funzionalità è detta binding dinamico (o late-binding), ed è supportato dai più diffusi linguaggi di programmazione ad oggetti.

Package

Il framework principale di java mette a disposizione moltissimi package e classi per l'uso più disparato. Il compito del programmatore non è conoscere tutto il framework e tutti i metodi di ogni classe, bensì avere un set di istruzioni ridotto ma consolidato.

In questo capitolo verranno presentate alcune classi più utilizzate e che meritano un'analisi specifica.

Package Util

Questo importante package contiene una vasta gamma di classi e interfacce, contiene inoltre uno dei sottosistemi più importanti di Java : il Collections Framework, una sofisticata gerarchia di interfacce e classi che forniscono tecnologia alla stato dell'arte per la gestione di gruppi di oggetti e che merita molta attenzione.

Di questo package molto esteso ci siamo limitati ad usare:

- ArrayList;
- Vector;
- Regex.

ArrayList

ArrayList è una classe generic con la seguente dichiarazione: class ArraList<E>.

In sostanza, un ArrayList è un array dinamico, in altre parole un ArrayList può aumentare o diminuire le proprie dimensioni in modo dinamico.

Vector

Vector implementa un array dinamico, è simile ad ArrayList ma con alcune differenze:

- è sincronizzata e contiene molti metodi legacy che duplicano le funzionalità dei metodi definiti dal Collection Framework;
- è stata riprogettata in modo da estendere AbstractList e implementare l'interfaccia List.

La dichiarazione è la seguente: class Vector<E>.

Regex

Il package java.util.regex è un sotto package di java.util, che supporta l'elaborazione di espressioni regolari. Un'espressione regolare è una stringa di caratteri che descrive una sequenza di caratteri. Tale descrizione generale, nota come schema, si può utilizzare per trovare corrispondenza in altre sequenze di caratteri. Le espressioni regolari possono specificare caratteri jolly, set di caratteri e diversi quantificatori.
Esistono due classi che lavorano insieme e supportano l'elaborazione di espressioni regolari:

- pattern;
- matcher.

Pattern serve per definire un'espressione regolare ed effettuare la corrispondenza dello schema utilizzando *Matcher*.

Package Sql

Il package java.Sql fornisce le API per l'accesso e l'elaborazione di dati memorizzati in un database. Questa API include diversi driver, nel nostro caso abbiamo usato JDBC.

Package IO

Il package java.IO fornisce supporto per le operazioni di I/O. In genere deve essere supportato da pacchetti esterni, come nel nostro caso i package per manipolare pdf e xml.

Package Awt

L'AWT contiene classi e metodi che consentono di creare e gestire finestre, nel nostro caso è stato utilizzato esclusivamente per la gestione degli eventi, poiché per l'interfaccia grafica abbiamo utilizza Swing.

Package Swing

Swing è un package Java, appartenente alle Java Foundation Classes (JFC) e orientato allo sviluppo di interfacce grafiche.
La libreria Swing viene utilizzata come libreria ufficiale per la realizzazione di interfacce grafiche in Java. È un'estensione del precedente Abstract Window Toolkit (AWT).

Strumenti software utilizzati

Per lo sviluppo dell'applicazione abbiamo utilizzato l'IDE *Eclipse*, quest'ultimo è un ambiente di sviluppo integrato scritto interamente in Java, offre quindi la possibilità di sviluppare su qualsiasi sistema operativo.
Eclipse fornisce supporto a diversi linguaggi di programmazione e offre anche dei plugin scaricabili dal marketplace che consentono di estendere le funzionalità dell'IDE stesso.

Per gestire il database sono stati utilizzati *MAMP* e *MySql Workbench*.

Per il "controllo versione" del codice abbiamo utilizzato GitHub che è un hosting per progetti software, permette di avere un controllo di versione distribuito.
Il vantaggio di usare questo software è la possibilità di tenere traccia di tutte le modifiche apportate al codice sorgente senza aver bisogno di un server centrale.
Con questo sistema gli sviluppatori possono lavorare parallelamente allo sviluppo dello stesso software, salvando le modifiche attraverso un *commit* ed effettuando successivamente la sincronizzazione con il progetto principale.
I commit e la sincronizzazione possono essere eseguiti tramite appositi comandi da terminale o, in alternativa, utilizzando la più semplice ed intuitiva interfaccia di "GitHub Desktop".

GitHub è completamente compatibile con Eclipse, quest'ultimo offre la possibilità di importare all'interno del proprio workspace un progetto presente sull'hosting.

Il deploy dell'applicazione è stato effettuato utilizzando Eclipse per la compilazione e creazione del file .jar ed in seguito il terminale per creare l'applicazione da installare sul proprio computer.

Il procedimento utilizzato, a partire dal jar, è il seguente:

PROCEDIMENTO DA TERMINALE

```
cd Desktop/circolo_cittadino

mkdir CircoloCittadino.iconset
sips -z 128 128 logo.png --out CircoloCittadino.iconset/icon_128x128.png
iconutil --convert icns CircoloCittadino.iconset

mkdir -p package/macosx
cp CircoloCittadino.icns package/macosx

jdk=$(/usr/libexec/java_home)
$jdk/bin/javapackager -version
$jdk/bin/javapackager -deploy -native dmg \
   -srcfiles circolo_cittadino.jar -appclass org.eclipse.jdt.internal.jarinjarloader.JarRsrcLoader -name CircoloCittadino \
   -srcfiles resources/ -name CircoloCittadino \
   -outdir deploy -outfile CircoloCittasino -v

cp deploy/bundles/circolo_cittadino-1.0.dmg circolo-cittadino-installer.dmg
```

Come primo passo è stata creata l'icona del nostro programma, in seguito è stata ridimensionata e convertita nel formato adatto.

Abbiamo poi creato la cartella che dovrà ospitare l'icona e copiato la stessa al suo interno. In seguito abbiamo usato dei comandi messi a disposizione dal jdk (Java Development Kit) per il deployment di un applicazione nativa, abbiamo selezionato i file che andranno a comporre l'applicazione e la directory nel quale salvare il file d'installazione.

Nel caso di windows per il deployment sono stati usati software di supporto:

- Launch4J per la creazione dell'exe;
- InnoSetupCompile per la creazione del setup.

Note finali

Il software è rilasciato con licenza Apache GNU GPL 3.0