



UNIVERSITÀ DI FIRENZE

Corso di Laurea in Ingegneria Informatica

INGEGNERIA DEL SOFTWARE

A.A 21/22

Leonardo Petrilli
Francesco Baiocchi

ChessApp

Indice

1	Introduzione	4
1.1	Statement	4
1.2	Motivazione	4
1.2.1	Dettagli tecnici	4
1.3	Strumenti utilizzati	5
2	Analisi dei requisiti	6
2.1	Use Case diagrams	6
2.1.1	Person-ChessPerson-User	6
2.1.2	Player	7
2.1.3	Tournament Manager	7
2.1.4	Referee	8
2.2	Use Case templates	9
2.3	Mockups	12
3	Progettazione	14
3.1	Class Diagram	14
3.1.1	Domain Model	14
3.1.2	Business Logic	15
3.2	Dettagli aggiuntivi	15
3.2.1	Observer	15
3.2.2	Factory	16
4	Classi ed Interfacce	18
4.1	Packaging	18
4.2	Domain Model	19
4.2.1	Tournament	19
4.2.2	Standings	20
4.2.3	Scoreboard	22
4.2.4	User	22
4.2.5	Player	23
4.2.6	Tournament Manager	23
4.2.7	Session	24
4.3	Business Logic	25

4.3.1	User Controller	25
4.3.2	Player Controller	25
4.3.3	Referee Controller	26
4.3.4	TournamentManager Controller	27
5	Testing	29
5.1	Test Session	29
5.2	Test Player	30
5.3	Test Tournament Manager	30
5.4	Test Referee	31
A		33
A.1	Che cos'è il punteggio Elo?	33
A.2	Swiss Pairing	34
A.3	ARO : Average Rating of Opponents	34
A.4	Metodo '400	34
B		35
B.1	Use Case Templates	35
C		36
C.1	Publish New Round	36

Capitolo 1

Introduzione

1.1 Statement

La nostra applicazione consente di visionare la lista dei prossimi tornei e per ognuno di questi è possibile consultare le rispettive informazioni generali (data di inizio/fine, lista dei giocatori, provincia, arbitro, ecc.). Un utente può inoltre creare un account di una delle seguenti tipologie :

- **Giocatore** → può iscriversi/disiscriversi a/da un torneo, visualizzare lo stato del torneo in corso (abbinamenti e classifiche);
- **Arbitro** → può caricare i risultati dei turni e di conseguenza aggiornare la classifica e il tabellone;
- **Organizzatore** → può creare un proprio torneo fornendo tutte le informazioni necessarie, gestire le iscrizioni.

1.2 Motivazione

Lo scopo di questo progetto è quello di realizzare un applicativo software per la gestione di tornei di scacchi. L'idea nasce dalla nostra passione comune per il gioco e dalle esperienze condivise durante gli anni.

Nonostante gli scacchi siano ampiamente diffusi a livello globale, ad oggi in Italia c'è una scarsa presenza di software per l'organizzazione di tornei.

1.2.1 Dettagli tecnici

Nel mondo degli scacchi la forza di un giocatore viene misurata attraverso un parametro numerico chiamato "*Punteggio Elo*". (vedere Appendice A.1)

Al termine di ogni partita questo parametro varia in base al risultato ottenuto (vittoria, pareggio o sconfitta) e al punteggio dell'avversario. L'obiettivo dei giocatori è quello di guadagnare quanti più punti Elo possibili partecipando a tornei.

Un torneo è costituito da più turni, in ognuno dei quali ogni giocatore si confronta con un avversario scelto attraverso un sistema di accoppiamento: il più diffuso tra questi è il Sistema Svizzero. Esso consiste nell'abbinare di volta in volta giocatori che abbiano accumulato un eguale numero di punti in classifica (1pt per vittoria, 0.5pt per pareggio, 0pt per sconfitta) oppure, in mancanza, giocatori con un punteggio vicino. (vedere Appendice A.2)

Al termine di ogni turno la classifica viene aggiornata in base ai punti accumulati da ogni partecipante e se necessario si utilizza un sistema di spareggio come il sistema Buchholz o il punteggio ARO. (vedere Appendice A.3)

1.3 Strumenti utilizzati

Analisi dei Requisiti & Progettazione :

- **StarUML** → Use Case Diagrams, Class Diagrams
- **Balsamiq** → Mockups

Implementazione :

- **Java** → Classi ed Interfacce
- **Eclipse** → IDE
- **JUnit** → Testing

Capitolo 2

Analisi dei requisiti

2.1 Use Case diagrams

Di seguito vengono presentati i diagrammi dei casi d'uso relativi agli attori del progetto.

2.1.1 Person-ChessPerson-User

Di seguito viene presentata la struttura gerarchica degli attori dell'applicazione. Al vertice della struttura vi è *Persona*, un'entità astratta nata per modellare i casi d'uso condivisi tra tutti gli attori. Casi d'uso di particolare importanza sono il Sign Up e il Login che permettono ad un Utente generico rispettivamente la creazione e l'utilizzo di account di vario tipo. (vedasi use case template a pag. 9 e 10)

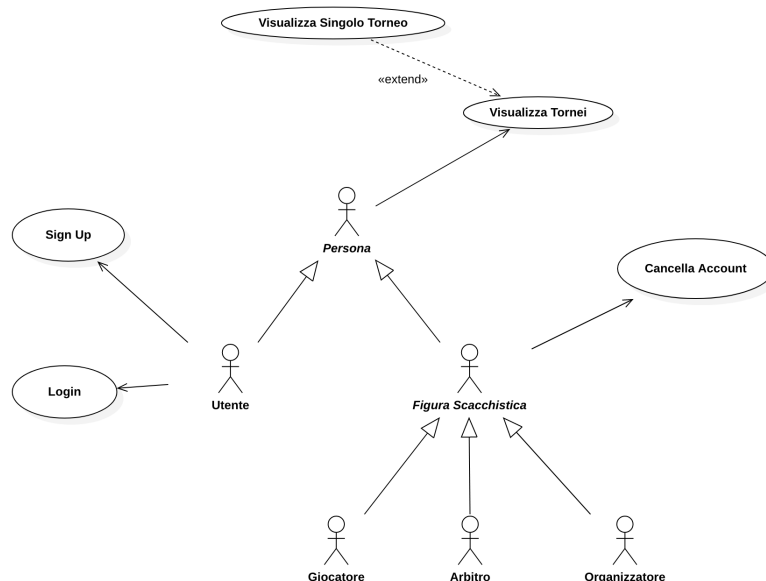


Figura 2.1: Struttura gerarchica degli attori

2.1.2 Player

Il caso d'uso fondamentale per questo attore è quello di consentire l'iscrizione ad un torneo. (per maggiori dettagli vedasi use case template a pag. 10)

Altri casi d'uso forniscono la possibilità di cancellare un'iscrizione, visualizzare le statistiche o il torneo in corso.

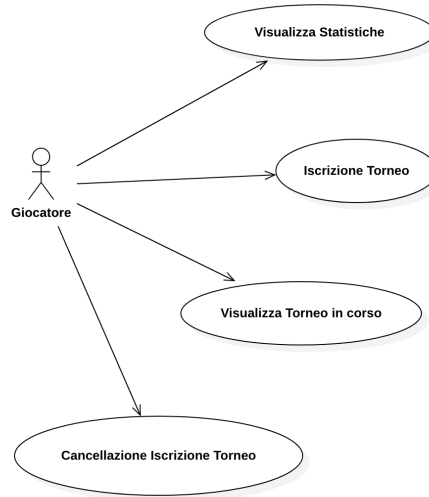


Figura 2.2: Use Case Diagram Giocatore

2.1.3 Tournament Manager

L'organizzatore è la figura chiave nella creazione e gestione di un torneo. Quando necessario egli può modificarne le informazioni generali (vedasi use case template a pag. 11) o, in casi estremi, cancellarlo. (vedasi use case template a pag. 11)

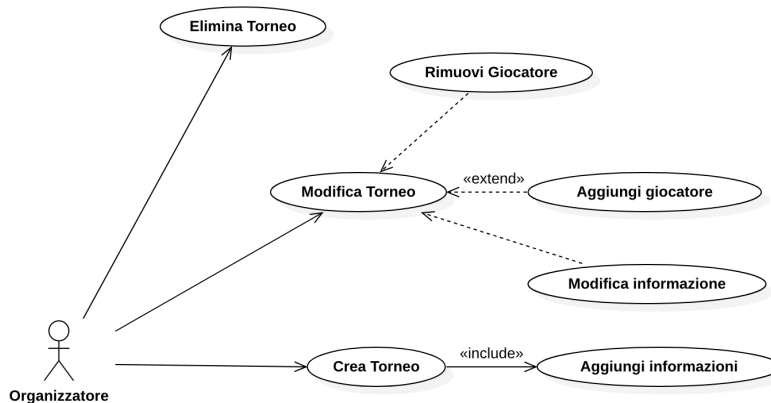


Figura 2.3: Use Case Diagram Organizzatore

2.1.4 Referee

L'arbitro si occupa della creazione/aggiornamento della classifica di un torneo (vedasi use case template a pag.12), del caricamento dei risultati e successivamente della pubblicazione dei nuovi turni.

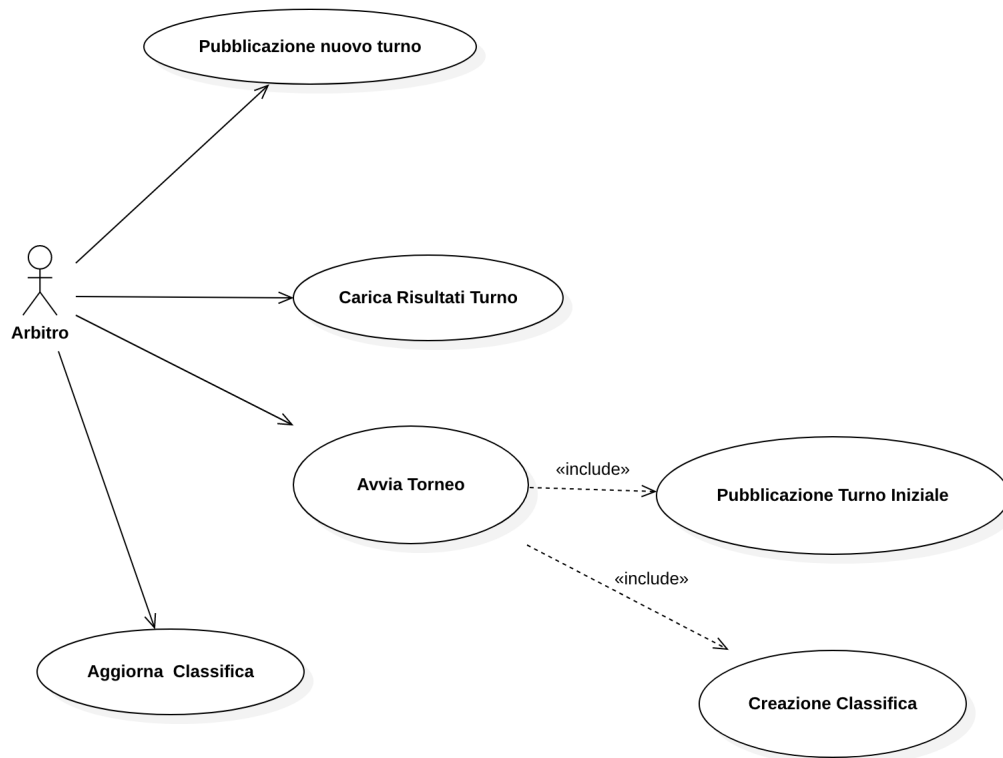


Figura 2.4: Use Case Diagram Arbitro

2.2 Use Case templates

Di seguito vengono approfonditi, attraverso l'uso di use case templates, gli use case più rilevanti per l'applicazione.

Sono presenti ulteriori use case templates nell'Appendice B.

Use-case field	Description
Use-Case Name :	Sign Up
Level :	User Goal
Actor :	User
Preconditions :	L'utente è nella pagina di SignUp.
Use-case overview :	L'utente inserisce nel form le informazioni per creare il suo account personale.
Normal flow :	<ol style="list-style-type: none">1. L'utente inserisce username e password per il suo profilo.2. L'utente sceglie la tipologia di account che vuole creare tra Arbitro, Giocatore o Manager. (vedasi Mockup a pag.12)3. L'utente clicca sul bottone di Sign Up.4. Il sistema mostra un form dove inserire tutti i dati richiesti per la creazione della tipologia di account scelto.5. Il sistema memorizza le informazioni dell'account creato.
Alternative flow :	<div>4.a Il sistema riconosce un username già utilizzato.</div> <div>5.a Il sistema mostra un messaggio di errore.</div>

Figura 2.5: Use Case Template Sign Up

Use-case field	Description
Use-Case Name :	Login
Level :	User Goal
Actor :	User
Preconditions :	L'utente è nella pagina di Login.
Use-case overview :	L'utente inserisce nel form le credenziali per loggarsi nel suo account personale.
Normal flow :	<ol style="list-style-type: none"> 1. L'utente inserisce nel form username e password. 2. L'utente clicca il bottone di Login.(vedasi Mockup a pag.13) 3. Il sistema riconosce come valide le credenziali inserite. 4. Il sistema mostra la pagina iniziale relativa al tipo di account dell'utente.
Alternative flow :	<ol style="list-style-type: none"> 3.a Il sistema riconosce che le credenziali inserite non sono valide. 4.a Il sistema mostra un messaggio di errore invitando l'utente a riprovare.

Figura 2.6: Use Case Template Login

Use-case field	Description
Use-Case Name :	Subscribe Tournament
Level :	User Goal
Actor :	Player
Preconditions :	Il giocatore è nella pagina iniziale.
Use-case overview :	Il giocatore si iscrive a un torneo.
Normal flow :	<ol style="list-style-type: none"> 1. Il giocatore seleziona il torneo dalla lista. 2. Il giocatore invia al sistema la richiesta di iscrizione. 3. Il sistema controlla che il giocatore non sia già iscritto al torneo. 4. Il sistema mostra la pagina contenente la lista dei partecipanti al torneo aggiornata.
Alternative flow :	<ol style="list-style-type: none"> 3.a Il sistema riconosce che il giocatore è già iscritto al torneo. 4.a Il sistema mostra un messaggio di errore.

Figura 2.7: Use Case Template Subscribe

Use-case field	Description
Use-Case Name :	Edit Information
Level :	User Goal
Actor :	Tournament Manager
Preconditions :	L'organizzatore è nella pagina contenente il catalogo dei tornei da lui organizzati.
Use-case overview :	L'organizzatore modifica un'informazione di un torneo.
Normal flow :	<ol style="list-style-type: none"> 1. L'organizzatore seleziona un torneo. 2. L'organizzatore sceglie quale informazione vuole modificare. 3. L'organizzatore sovrascrive l'informazione. 4. Il sistema mostra una pagina con le informazioni del torneo aggiornate.

Figura 2.8: Use Case Template EditInformation

Use-case field	Description
Use-Case Name :	Remove Tournament
Level :	User Goal
Actor :	Tournament Manager
Preconditions :	L'organizzatore è nella pagina contenente il catalogo dei tornei da lui organizzati.
Use-case overview :	L'organizzatore rimuove un torneo dal catalogo.
Normal flow :	<ol style="list-style-type: none"> 1. L'organizzatore seleziona il torneo che vuole rimuovere. 2. L'organizzatore invia una richiesta di eliminazione del torneo al sistema. 3. Il sistema elimina il torneo dal catalogo. 4. Il sistema mostra una pagina con il catalogo aggiornato.

Figura 2.9: Use Case Template Remove Tournament

Use-case field	Description
Use-Case Name :	Update Standings
Level :	User Goal
Actor :	Referee
Preconditions :	L'arbitro ha caricato i risultati del turno precedente.
Use-case overview :	L'arbitro aggiorna la classifica del torneo tenendo conto dei risultati del turno precedente.
Normal flow :	<ol style="list-style-type: none"> 1. L'arbitro richiede al sistema di aggiornare la classifica del torneo. 2. Il sistema calcola la nuova classifica. 3. Il sistema mostra una pagina contenente la classifica aggiornata.
Alternative flow :	<ol style="list-style-type: none"> 4.a Il sistema riconosce un username già utilizzato. 5.a Il sistema mostra un messaggio di errore.

Figura 2.10: Use Case Template Update Standings

2.3 Mockups

Di seguito vengono presentati alcuni mockups relativi ad una possibile implementazione dell'interfaccia grafica dell'applicazione.

ChessAppSignUpPage

 **ChessApp**

Username

Password

Choose the type of the account...

Figura 2.11: Mockup SignUp page

ChessAppLoginPage



ChessApp


Username

Password

Don't have an account? **Sign Up!**

Figura 2.12: Mockup Login page

ChessAppHomePage



ChessApp

Torneo	Regione	Provincia	Inizio	Fine	
5° Riviera dei Cedri	Calabria	CS	06-09-2010	12-09-2010	
1° Festival città di Riposto	Sicilia	CT	08-03-2010	16-03-2010	
8° Campionato Regionale	Calabria	CT	09-10-2010	17-10-2010	
Week-End di Primavera	Lazio	RM	01-04-2011	03-04-2011	
Internazionale Livorno	Toscana	LI	01-07-2011	03-07-2011	
Semilampo Rocca Priora	Lazio	RM	01-10-2011	01-10-2011	
2° Memorial Lucin	Trentino	TN	06-11-2011	09-11-2011	
Torneo Round-Robin	Toscana	PO	12-11-2012	15-11-2022	
X° TORNEO "Città di Morrovalle"	Marche	MC	12-08-2012	15-08-2010	

Figura 2.13: Mockup Main page

3.1.2 Business Logic

In questo package sono presenti i controller che contengono i metodi corrispondenti ai casi d'uso elencati in precedenza.

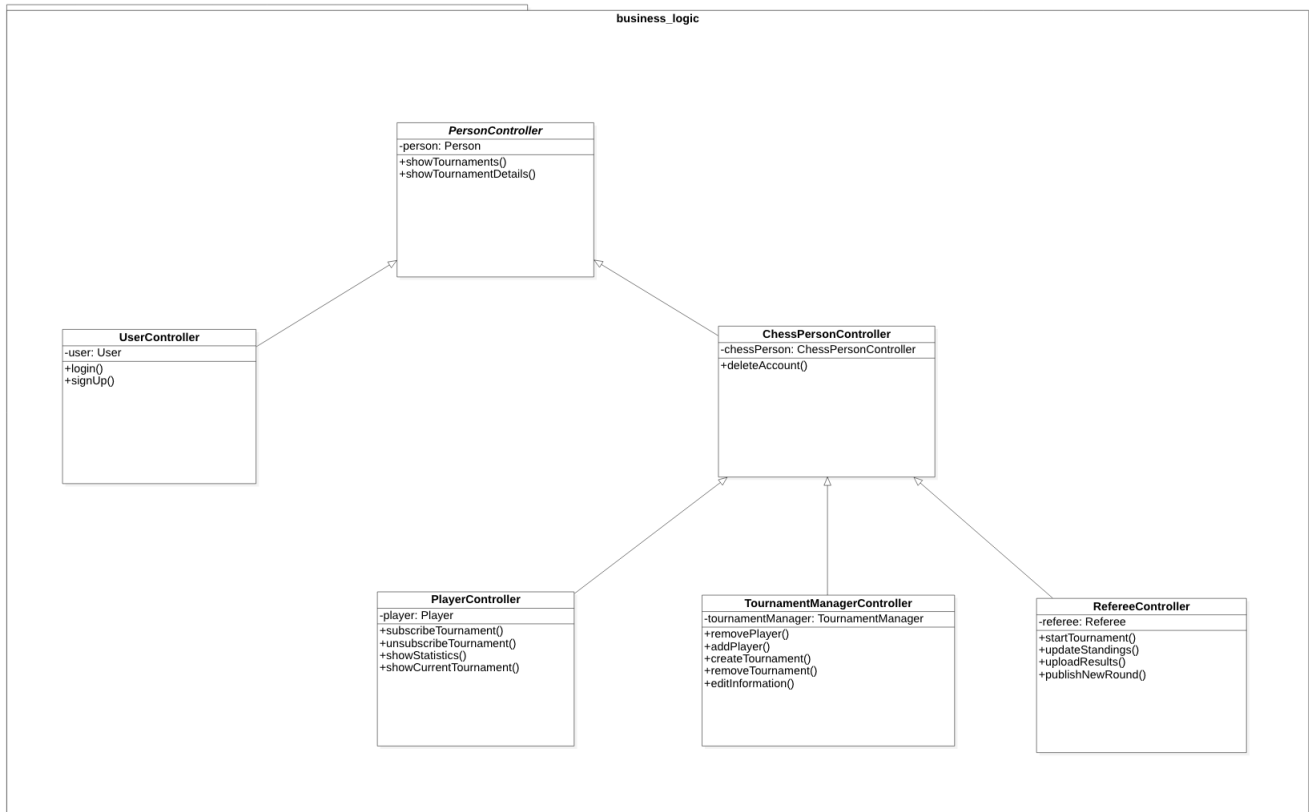


Figura 3.2: Class Diagram Business Logic

3.2 Dettagli aggiuntivi

3.2.1 Observer

Il design pattern Observer viene utilizzato per gestire l'aggiornamento del Global Tournament Catalog, classe che è responsabile della visualizzazione della lista complessiva dei tornei. Questo avviene nel momento in cui un Tournament Manager decide di aggiungere o rimuovere un torneo dal Manager Tournament Catalog, ossia il suo catalogo personale.

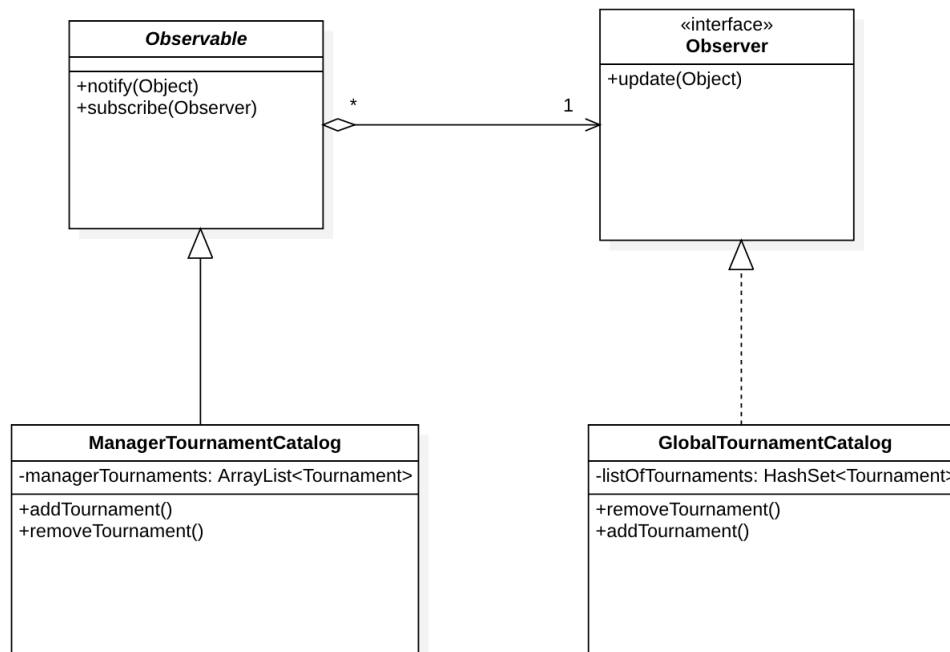


Figura 3.3: Design Pattern Observer

Come si evince dall'immagine, il pattern è stato implementato in versione *push*. La motivazione risiede nella presenza di un solo tipo di Observer e quindi non c'è necessità per l'Observable di notificare l'intero stato. Nel nostro caso infatti, il Global Tournament Catalog per l'aggiornamento ha bisogno soltanto del torneo che è stato aggiunto/rimosso e non del catalogo di tutti i tornei del Manager. In questo modo viene garantito un accoppiamento lasco tra managerTournamentCatalog e GlobalTournamentCatalog in quanto quest'ultimo non ha bisogno di conoscere come è implementato managerTournamentCatalog, non ottenendo mai un suo riferimento.

3.2.2 Factory

Il design pattern Factory viene utilizzato per la creazione dinamica di differenti tipologie di account (Arbitro, Giocatore, Organizzatore) da parte di un utente. Quest'ultimo utilizza il metodo *createActor()* della UserFactory comunicando il tipo di account e i dati necessari per la sua creazione.

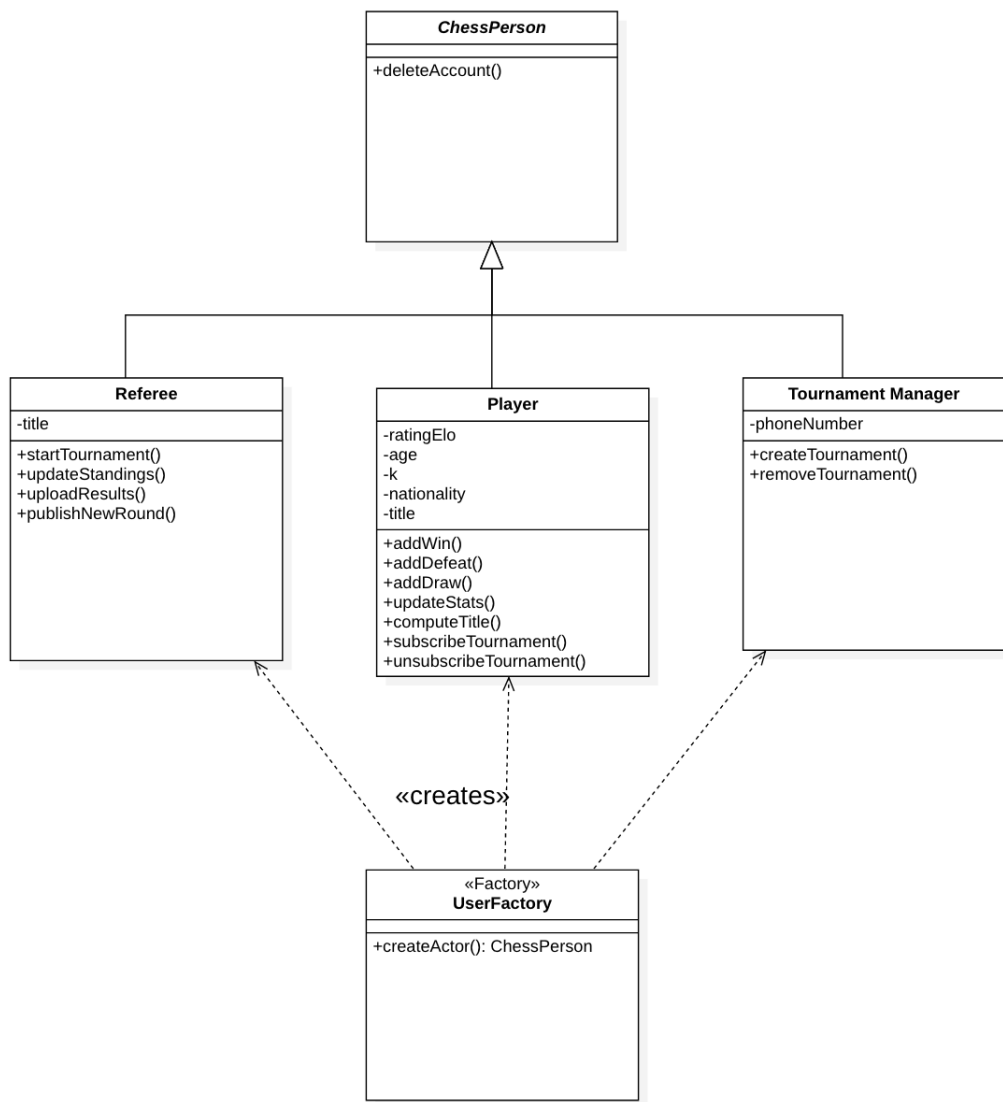


Figura 3.4: Design Pattern Factory

Tale pattern permette all'applicazione di essere *aperta* all'aggiunta di nuove tipologie di account, il linea con il principio di aperto/chiuso.

Capitolo 4

Classi ed Interfacce

Viene presentata di seguito l'implementazione delle classi del progetto che meritano un approfondimento e la divisione in package.

4.1 Packaging



Figura 4.1: Divisione in Package

4.2 Domain Model

4.2.1 Tournament

Questa classe mantiene tutte le informazioni relative ad un torneo, come la classifica e il tabellone degli abbinamenti.

```
public class Tournament {  
    private String name;  
    private String province;  
    private String region;  
    private String startingDate;  
    private String endingDate;  
    private String timeControl;  
    private int numOfRounds;  
    private String refereeName;  
    private String managerName;  
    private boolean started = false;
```

I suoi metodi consentono l'avvio (metodo *start()*) e il corretto svolgimento del torneo. Dopo il caricamento dei risultati da parte dell'arbitro (*uploadResults()*), avviene l'aggiornamento della classifica e del tabellone (*updateStandings()* e *publishNewRound()*).

```
    public void updateStandings() {  
        standings.updateStandings(new Scoreboard(scoreboard));  
    }  
  
    public void uploadResults(String[] results) {  
        scoreboard.uploadResults(results, getListOfPlayers());  
    }  
  
    public void publishNewRound() throws Exception {  
        if ((standings.getCurrentRound()-1) != numOfRounds) {  
            scoreboard.publishNewRound(getStandings());  
        }else {  
            throw new Exception("Tournament finished");  
        }  
    }  
  
    public void start() {  
        standings = new Standings(listOfPlayers.size());  
        standings.initialize(getListOfPlayers());  
  
        scoreboard = new Scoreboard(listOfPlayers.size());  
        scoreboard.initialize(getStandings());  
  
        for (int i = 0; i < listOfPlayers.size(); i++) {  
            listOfPlayers.get(i).setPlayingTournament(this);  
        }  
    }  
}
```

4.2.2 Standings

Questa classe mantiene tutte le informazioni dei giocatori utili a rappresentare la classifica di un torneo, come ad esempio i punti accumulati (PTS), l'Elo, l'ARO, ecc...

```
public class Standings {  
  
    private int numOfPlayers;  
    private String[][] standings;  
    private int currentRound;  
  
    public Standings(int numOfPlayers) {  
  
        this.numOfPlayers = numOfPlayers;  
        this.currentRound = 0;  
        standings = new String[numOfPlayers + 1][8];  
        standings[0][0] = "POS";  
        standings[0][1] = "PTS";  
        standings[0][2] = "TITLE";  
        standings[0][3] = "NAME";  
        standings[0][4] = "ELO";  
        standings[0][5] = "PERF";  
        standings[0][6] = "NAT";  
        standings[0][7] = "ARO";  
  
    }  
}
```

Nella prossima pagina viene presentato il metodo più rilevante della classe : *updateStandings()*, il quale si occupa di aggiornare i campi della classifica e di mantenerla ordinata. L'ordinamento avviene sulla base del nuovo punteggio in classifica ed è implementato grazie all'utilizzo di comparatori (libreria *java.util*), la quale logica prevede anche un sistema di spareggio con ARO e Elo.

```
public void sortStandings() {  
  
    ArrayList<CriteriaStandings> criteria = new ArrayList<CriteriaStandings>();  
  
    for(int i=1;i<numOfPlayers + 1;i++) {  
  
        criteria.add(new CriteriaStandings(Integer.parseInt(standings[i][0]),Float.parseFloat(standings[i][1]),  
            Float.parseFloat(standings[i][7]),Integer.parseInt(standings[i][4])));  
  
    }  
  
    criteria.sort(new CriteriaComparator());  
    String[][] sortedStandings = new String[numOfPlayers+1][8];  
    sortedStandings[0] = standings[0];  
    for(int i=1;i<numOfPlayers+1;i++) {  
        sortedStandings[i] = standings[criteria.get(i-1).getPos()];  
    }  
  
    standings = sortedStandings;  
}
```

```

public void updateStandings(Scoreboard scoreboard) {

    for(int i=1; i < (int) Math.ceil((double)numOfPlayers/2)+ 1; i++) {
        for(int j = 1; j < numOfPlayers + 1; j++) {

            float tmp = Float.parseFloat(standings[j][1]);
            float aro = Float.parseFloat(standings[j][7]);
            int opponentElo = 0;
            int performance = Integer.parseInt(standings[j][5]);

            if(scoreboard.getElement(i,1).equals(standings[j][3])) {

                for(int k=1; k < numOfPlayers + 1; k++) {
                    if(scoreboard.getElement(i, 2).equals(standings[k][3])) {
                        opponentElo = Integer.parseInt(standings[k][4]);
                    }
                }

                aro = (aro * currentRound + opponentElo)/(currentRound+1);
                standings[j][7] = Float.toString(aros);

                if(scoreboard.getElement(i, 3).charAt(1) == '.') {
                    tmp += 0.5;
                    standings[j][1] = Float.toString(tmp);
                    performance = Math.round((float)(performance * currentRound + opponentElo)/(currentRound+1));
                }else {
                    tmp += Float.parseFloat(scoreboard.getElement(i, 3).substring(0, 1));
                    standings[j][1] = Float.toString(tmp);

                    if(scoreboard.getElement(i,3).charAt(0) == '1') {
                        performance = Math.round((float)(performance * currentRound + opponentElo + 400)/(currentRound+1));
                    }else {
                        performance = Math.round((float)(performance * currentRound + opponentElo - 400)/(currentRound+1));
                    }
                }

                standings[j][5] = Integer.toString(performance);
            }

        }else if(scoreboard.getElement(i, 2).equals(standings[j][3])){

            for(int k=1; k < numOfPlayers + 1; k++) {
                if(scoreboard.getElement(i, 1).equals(standings[k][3])) {
                    opponentElo = Integer.parseInt(standings[k][4]);
                }
            }

            aro = (aro * currentRound + opponentElo)/(currentRound+1);
            standings[j][7] = Float.toString(aros);

            if(scoreboard.getElement(i, 3).charAt(1) == '.') {
                tmp += 0.5;
                standings[j][1] = Float.toString(tmp);
                performance = Math.round((float)(performance * currentRound + opponentElo)/(currentRound+1));
            }else {
                tmp += Float.parseFloat(scoreboard.getElement(i, 3).substring(2));
                standings[j][1] = Float.toString(tmp);
                if(scoreboard.getElement(i,3).charAt(2) == '1') {
                    performance = Math.round((float)(performance * currentRound + opponentElo + 400)/(currentRound+1));
                }else {
                    performance = Math.round((float)(performance * currentRound + opponentElo - 400)/(currentRound+1));
                }
            }

            standings[j][5] = Integer.toString(performance);
        }
    }

    currentRound++;
    sortStandings();
    for(int i=1; i < numOfPlayers +1; i++) {
        standings[i][0] = Integer.toString(i);
    }
}

```

4.2.3 Scoreboard

Questa classe mantiene gli abbinamenti del turno corrente ed i risultati delle partite pubblicati dall'arbitro. Per ogni giocatore vengono anche memorizzate le informazioni relative ai turni precedenti, come i colori (bianco/nero) e gli avversari contro cui ha giocato.

```
public class Scoreboard {
    private int numOfPlayers;
    private String[][] board;
    private Map<String, ArrayList<String>> previousPairings = new HashMap<String, ArrayList<String>>();
    private Map<String, String> previousColour = new HashMap<String, String>();
    private Map<String, Float> eloVariations = new HashMap<String, Float>();

    public Scoreboard(int numOfPlayers) {
        this.numOfPlayers = numOfPlayers;
        board = new String[(((int) Math.ceil((double) numOfPlayers / 2)) + 1)][4];
        board[0][0] = "Table";
        board[0][1] = "White";
        board[0][2] = "Black";
        board[0][3] = "Result";
    }
}
```

Il metodo più importante di questa classe è il *publishNewRound()* che, utilizzando le informazioni sopra citate, crea gli abbinamenti del nuovo turno sulla base del sistema di accoppiamento svizzero.

Per ragioni di spazio il codice del metodo è presente nell'Appendice C.

4.2.4 User

Questa classe modella l'entità utente ovvero una qualsiasi persona interessata alla creazione di un account o, se già in possesso di questo, all'accesso mediante credenziali (username e password).

```
public class User extends Person {
    private AccountCreator accountCreator;

    public User(String name, String surname, TournamentDisplay tournamentDisplay, AccountCreator accountCreator) {
        super(name, surname, tournamentDisplay);
        this.accountCreator = accountCreator;
    }

    public void createAccount(Object[] params, UserType type, String username, String password)
        throws IllegalArgumentException {
        accountCreator.createAccount(params, type, username, password);
    }

    public ChessPerson signIn(String username, String password) throws IllegalArgumentException {
        return accountCreator.signIn(username, password);
    }
}
```

4.2.5 Player

La classe `Player` contiene i dati relativi all'anagrafica di un giocatore (nome, cognome, età, nazionalità) così come indici di abilità (punteggio Elo, k). Associato ad ogni giocatore vi è un insieme di statistiche (*playerStats*) che descrivono le performance negli ultimi tornei.

```
public class Player extends ChessPerson {  
    private int age;  
    private int ratingElo;  
    private String title;  
    private int k;  
    private String nationality;  
  
    private ArrayList<Tournament> subscribedTournaments = new ArrayList<Tournament>();  
    private Tournament playingTournament = null;  
    private Stats playerStats;
```

Il metodo *updateStats()* si occupa di aggiornare le statistiche sopra citate al termine di ogni torneo.

```
public void updateStats() {  
    playerStats.setNumOfGames(playerStats.getNumOfGames() + playingTournament.getNumOfRounds());  
    playerStats.setNumOfTournaments(playerStats.getNumOfTournaments() + 1);  
    playerStats.setLastPerformance(playingTournament.getPerformance(getName() + " " + getSurname()));  
  
    float playerScore = playingTournament.getPlayerScore(this);  
    float playerVariation = playingTournament.getPlayerVariation(this);  
  
    ratingElo += (playerScore - playerVariation) * k;  
  
    if (ratingElo < playerStats.getLowestRatingElo()) {  
        playerStats.setLowestRatingElo(ratingElo);  
    } else if (ratingElo > playerStats.getHighestRatingElo()) {  
        playerStats.setHighestRatingElo(ratingElo);  
    }  
    title = computeTitle(playerStats.getHighestRatingElo());  
    playingTournament = null;  
}
```

4.2.6 Tournament Manager

Tournament Manager è la classe che modella la figura dell'organizzatore, ossia di colui che ha i privilegi esclusivi di creare e cancellare tornei. Inoltre ogni organizzatore possiede un proprio catalogo personale dove tiene traccia dei tornei che sta gestendo.

```
public class TournamentManager extends ChessPerson {  
    private String phoneNumber;  
    private ManagerTournamentCatalog managerCatalog;  
  
    public void addTournament(Tournament tournament) {  
        managerCatalog.addTournament(tournament);  
    }
```

4.2.7 Session

Questa classe gestisce gli account implementando le interfacce *AccountCreator* e *AccountRemover*.

L'attributo *accounts* tiene traccia delle credenziali permettendo così di controllarne la correttezza al momento del Login.

L'attributo *userMap* è utilizzato invece per memorizzare tutte le informazioni relative ad un account.

```
public class Session implements AccountCreator, AccountRemover {  
    private Map<String, String> accounts = new HashMap<String, String>();  
    private Map<String, ChessPerson> userMap = new HashMap<String, ChessPerson>();  
    private UserFactory userFactory = new UserFactory();
```

Il metodo *createAccount()* crea un account del tipo specificato da *type* attraverso l'utilizzo del metodo *createActor()* della *UserFactory* verificando anche che lo username non sia già stato scelto per un altro profilo.

Il metodo *signIn()* svolge in maniera simile a *createAccount()* un controllo sulle credenziali inserite e, se corrette, restituisce un riferimento alla classe base astratta *ChessPerson*. Esso, grazie al polimorfismo, contiene le informazioni relative ad un tipo di account. (*Player*, *Referee*, *TournamentManager*)

```
public void createAccount(Object[] params, UserType type, String username, String password)  
    throws IllegalArgumentException {  
    ChessPerson account = userFactory.createActor(params, type);  
    if (accounts.containsKey(username)) {  
        throw new IllegalArgumentException("Username already exists.");  
    }  
    accounts.put(username, password);  
    userMap.put(username, account);  
}  
  
public ChessPerson signIn(String username, String password) throws IllegalArgumentException {  
    if (accounts.containsKey(username)) {  
        if (accounts.get(username).equals(password)) {  
            return userMap.get(username);  
        } else {  
            throw new IllegalArgumentException("Wrong Password.");  
        }  
    } else {  
        throw new IllegalArgumentException("Username doesn't exist.");  
    }  
}
```


4.3 Business Logic

4.3.1 User Controller

Tramite questo controller lo User può creare un nuovo account utilizzando il metodo *SignUp()*, oppure loggarsi al suo account con il metodo *login()* fornendo le proprie credenziali.

```
public class UserController extends PersonController {  
    private User user;  
    public UserController(User user) {  
        super(user);  
        this.user = user;  
    }  
    public ChessPerson login(String username, String password) {  
        try {  
            return user.signIn(username, password);  
        } catch (IllegalArgumentException e) {  
            System.out.println(e.getMessage());  
            return null;  
        }  
    }  
    public void signUp(Object[] params, UserType type, String username, String password) {  
        try {  
            user.createAccount(params, type, username, password);  
        } catch (IllegalArgumentException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Al momento del *SignUp()* nel caso in cui venga fornito uno username di un account già esistente viene gestita l'eccezione lanciata nella classe Session. Accade lo stesso all'interno del metodo *login()* quando vengono fornite credenziali non valide.

4.3.2 Player Controller

È un controller che permette al Player di partecipare a tornei, visualizzare le proprie statistiche e il torneo in corso.

```

public class PlayerController extends ChessPersonController {

    private Player player;

    public PlayerController(Player player) {
        super(player);
        this.player = player;
    }

    public void subscribeTournament(Tournament t) {
        try {
            if (!(t.isStarted())) {
                player.subscribeTournament(t);
                t.addPlayer(player);
            } else {
                System.out.println("Tournament already started.");
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public void unsubscribeTournament(Tournament t) {
        if (!(t.isStarted())) {
            player.unsubscribeTournament(t);
            t.removePlayer(player);
        } else {
            System.out.println("Tournament already started.");
        }
    }

    public Stats showStatistics() {
        return player.getStatistics();
    }

    public Tournament showCurrentTournament() {
        try {
            return player.getCurrentTournament();
        } catch (Exception e) {
            System.out.println(e.getMessage());
            return null;
        }
    }
}

```

Nel metodo *subscribeTournament()* viene gestito il caso in cui un Player prova a iscriversi ad un torneo già iniziato, questo avviene attraverso l'utilizzo del flag *started* della classe *Tournament*. In maniera simile tale flag viene utilizzato nel metodo *unsubscribeTournament()* per evitare che un Player possa disiscriversi da un torneo che sta giocando.

4.3.3 Referee Controller

Questo controller realizza i casi d'uso del Referee ed è quindi di fondamentale importanza per l'avvio e il corretto svolgimento di un torneo. Si noti come nel metodo *publishNewRound()* viene gestito il caso in cui il Referee provi a pubblicare un nuovo turno dopo la fine del torneo.

```

public class RefereeController extends ChessPersonController {

    private Referee referee;

    public RefereeController(Referee referee) {
        super(referee);
        this.referee = referee;
    }

    public void startTournament() {
        referee.startTournament();
        referee.getActiveTournament().setStarted(true);
    }

    public void updateStandings() {
        referee.updateStandings();
        if(referee.getActiveTournament().getNumOfRounds() == referee.getActiveTournament().getStandings().getCurrentRound())
            for (int i = 0; i < referee.getActiveTournament().getListOfPlayers().size(); i++) {
                referee.getActiveTournament().getListOfPlayers().get(i).updateStats();
            }
    }

    public void uploadResults(String[] results) {
        referee.uploadResults(results);
    }

    public void publishNewRound() {
        try {
            referee.publishNewRound();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}

```

Ciascun metodo in figura rispecchia una fase dello svolgimento di un torneo a partire dall'avvio di questo (*startTournament()*), in cui si inizializzano classifica e tabellone, il caricamento dei risultati (*uploadResults()*), l'aggiornamento della classifica (*updateStandings()*) e infine la pubblicazione del nuovo turno *publishNewRound()*.

4.3.4 TournamentManager Controller

È il controller che permette al TournamentManager (organizzatore del torneo) di creare, cancellare un torneo o modificarne le informazioni.

Nello specifico il metodo *editInformation()* implementa l'operazione di modifica ricevendo in ingresso il torneo *t* contenente l'informazione *nameInformation* da modificare con il nuovo dato *information*. (vedasi la figura alla prossima pagina)

```

public class TournamentManagerController extends ChessPersonController {
    private TournamentManager tournamentManager;

    public TournamentManagerController(TournamentManager tournamentManager) {
        super(tournamentManager);
        this.tournamentManager = tournamentManager;
    }

    public Tournament createTournament(String name, String province, String region, String startingDate, String endingDate,
        String timeControl, int numOfRounds, String refereeName, String managerName) {
        Tournament t = new Tournament(name, province, region, startingDate, endingDate, timeControl, numOfRounds,
            refereeName, tournamentManager.getName());
        tournamentManager.addTournament(t);
        return t;
    }

    public void addPlayer(Tournament t, Player p) {
        try {
            t.addPlayer(p);
            p.subscribeTournament(t);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public void editInformation(Tournament t, String nameInformation, Object information) {
        switch (nameInformation) {
            case "name":
                t.setName((String) information);
                break;
            case "province":
                t.setProvince((String) information);
                break;
            case "region":
                t.setRegion((String) information);
                break;
            case "startingDate":
                t.setStartingDate((String) information);
                break;
            case "endingDate":
                t.setEndingDate((String) information);
                break;
            case "timeControl":
                t.setTimeControl((String) information);
                break;
            case "numOfRounds":
                t.setNumOfRounds((int) information);
                break;
            case "refereeName":
                t.setRefereeName((String) information);
                break;
        }
    }
}

```

Per ulteriori approfondimenti o curiosità sui dettagli implementativi dell'applicazione è possibile accedere al codice completo nella repo Github al seguente link <https://github.com/francescobaiio/SWE-ChessApp>.

Capitolo 5

Testing

5.1 Test Session

Di seguito si riporta le *suite* di test finalizzata a verificare il corretto funzionamento dei casi d'uso di *signUp()*, *login()* e di *deleteAccount()*.

```
public class SessionTest {

    private static PlayerController playerController;
    private static TournamentManagerController tournamentManagerController;
    private static RefereeController refereeController;
    private static UserController userController;
    private static Session session;
    private static User user;
    private static GlobalTournamentCatalog tournamentCatalog;
    private static ManagerTournamentCatalog managerTournamentCatalog;

    @BeforeAll
    static void setUpBeforeClass() throws Exception {

        tournamentCatalog = new GlobalTournamentCatalog();
        session = new Session();
        user = new User("Mario", "Rossi", tournamentCatalog, session);
        userController = new UserController(user);
        managerTournamentCatalog = new ManagerTournamentCatalog(tournamentCatalog);
    }

    @Test
    public void loginPlayerTest() {

        Object[] userParams = {"Mario", "Rossi", tournamentCatalog, session, 1850, 21, "1N", "ITA"};
        userController.signUp(userParams, UserType.Player, "marietto01", "BobbyFischer21");
        assertNotNull(session.getUserMap().get("marietto01"));
        assertNotNull(session.getAccounts().get("marietto01"));
        assertEquals("BobbyFischer21", session.getAccounts().get("marietto01"));

        Player p = (Player) userController.login("marietto01", "BobbyFischer21");
        playerController = new PlayerController(p);
        assertNotNull(p);

        playerController.deleteAccount("marietto01", "BobbyFischer21");
        assertNull(session.getAccounts().get("marietto01"));
        assertNull(session.getUserMap().get("marietto01"));
    }
}
```

Per brevità è stato riportato solamente il *loginPlayerTest()* tuttavia sono stati realizzati anche i test, sugli stessi casi d'uso, relativi agli altri tipi di account: Referee e TournamentManager.

5.2 Test Player

I test in questa suite hanno lo scopo di verificare il funzionamento delle operazioni principali della classe Player.

```
public class PlayerTest {

    private static PlayerController playerController;
    private static Player player;
    private static Tournament tournament;
    private static Referee referee;
    private static RefereeController refereeController;

    @BeforeAll
    public static void setUpBeforeClass() throws Exception {

        player = new Player("Francesco", "Beccarini", null, null, 2130, 21, "CM", "ITA");
        playerController = new PlayerController(player);
        tournament = new Tournament("Campionato Provinciale Rieti", "RI", "Lazio", "31/08/2022", "06/09/2022",
            9, "Fabrizio Falsi", "Erminio Castaldi");
        referee = new Referee("Fabrizio", "Falsi", null, null, tournament, "AN");
        refereeController = new RefereeController(referee);

    }

    @Test
    public void testSubscribe() {

        playerController.subscribeTournament(tournament);
        assertEquals(tournament, player.getSubscribedTournaments().get(0));
        assertEquals(player, tournament.getListOfPlayers().get(0));

        playerController.unsubscribeTournament(tournament);
        assertEquals(0, player.getSubscribedTournaments().size());
        assertEquals(0, tournament.getListOfPlayers().size());

    }

    @Test
    public void testShowCurrentTournament() {

        playerController.subscribeTournament(tournament);
        refereeController.startTournament();
        assertNotNull(playerController.showCurrentTournament());
        tournament.stopTournament();
        playerController.unsubscribeTournament(tournament);

    }

}
```

Nello specifico attraverso *testSubscribe()* viene comprovata la correttezza dei casi d'uso corrispondenti a iscrizione e disiscrizione di un Player ad un torneo. Invece in *testShowCurrentTournament()* viene assicurato il fatto che un Player visualizzi senza errori il torneo che sta giocando in quel momento.

5.3 Test Tournament Manager

Questa test suite permette di convalidare il funzionamento dei casi d'uso del Tournament Manager. Nel *setUp()* vengono istanziati tutti gli oggetti necessari ai successivi test case, come alcuni giocatori, alcuni tornei, un organizzatore e il suo relativo catalogo.

```

public class TournamentManagerTest {

    private static TournamentManagerController tournamentManagerController;
    private static TournamentManager tournamentManager;
    private static ManagerTournamentCatalog managerTournamentCatalog;
    private static GlobalTournamentCatalog tournamentCatalog;
    private static Player p1, p2;
    private static Tournament t1, t2;

    @BeforeAll
    public static void setUpBeforeClass() throws Exception {

        tournamentCatalog = new GlobalTournamentCatalog();
        managerTournamentCatalog = new ManagerTournamentCatalog(tournamentCatalog);
        p1 = new Player("Francesco", "Beccarini", tournamentCatalog, null, 2130, 21, "CM", "ITA");
        p2 = new Player("Cristiano", "Berrettoni", tournamentCatalog, null, 1759, 21, "2N", "ITA");

        tournamentManager = new TournamentManager("Erminio", "Castaldi", tournamentCatalog, null, "3928154921",
            managerTournamentCatalog);
        tournamentManagerController = new TournamentManagerController(tournamentManager);
        t1 = new Tournament("Campionato Provinciale Roma", "RM", "Lazio", "11/05/2021", "16/05/2021", "90+30", 7,
            "Mario Rossi", "Gino Bianchi");
        managerTournamentCatalog.addTournament(t1);
        t1.addPlayer(p1);
        t2 = new Tournament("Campionato Provinciale Virtebo", "VI", "Lazio", "11/05/2021", "16/05/2021", "90+30", 7,
            "Luigi Verdi", "Eugenio Latorre");
        managerTournamentCatalog.addTournament(t2);
    }
}

```

Qui sotto in figura viene riportato il codice dei vari test. Si noti come in questo caso ciascuna funzionalità venga testata singolarmente non essendoci vincoli di sequenzialità tra i casi d'uso.

```

@Test
public void testCreateTournament() throws Exception {

    Tournament t = tournamentManagerController.createTournament("Campionato Provinciale Rieti", "RI", "Lazio",
        "31/08/2022", "06/09/2022", "90+30", 9, "Fabrizio Falsi", "Erminio Castaldi");

    assertNotNull(t);
    assertTrue(managerTournamentCatalog.getManagerTournaments().contains(t));
    assertTrue(p1.showTournaments().contains("Campionato Provinciale Rieti"));
    assertEquals(t, p1.showTournamentDetails("Campionato Provinciale Rieti"));
}

@Test
public void testDeleteTournament() {

    tournamentManagerController.removeTournament(t1);
    assertFalse(managerTournamentCatalog.getManagerTournaments().contains(t1));
    assertFalse(p1.showTournaments().contains("Campionato Provinciale Roma"));
}

@Test
public void testAddPlayer() {

    tournamentManagerController.addPlayer(t2, p2);
    assertTrue(t2.getListOfPlayers().contains(p2));
    assertTrue(p2.getSubscribedTournaments().contains(t2));
}

@Test
public void testRemovePlayer() {

    tournamentManagerController.removePlayer(t2, p1);
    assertFalse(t2.getListOfPlayers().contains(p1));
    assertFalse(p1.getSubscribedTournaments().contains(t2));
}

@Test
public void testEditInformation() {

    tournamentManagerController.editInformation(t1, "startingDate", "10/05/2021");
    assertTrue(t1.getStartingDate().equals("10/05/2021"));
    tournamentManagerController.editInformation(t1, "numOfRounds", 9);
    assertEquals(9, t1.getNumOfRounds());
}
}

```

5.4 Test Referee

Questa test suite verifica i casi d'uso dell'arbitro e quindi, in maniera completa, lo svolgimento di un torneo.

Non viene riportato il *set up* della suite poichè consiste nelle semplici e numerose istanziazioni di giocatori, organizzatori, arbitri e tornei.

Qui di seguito viene riportato un **frammento** di codice relativo al test case *testEvenPlayersTournament()*. Questo verifica il corretto avvio del torneo, il caricamento dei risultati, l'aggiornamento della classifica e la pubblicazione del nuovo turno.

```
@Test
public void testEvenPlayersTournament() {

    String[][] initialRoundStandings = { { "1", "0", "1N", "Leonardo Petrilli", "1922", "1922", "ITA", "0" },
    { "2", "0", "1N", "Francesco Giovanzanti", "1792", "1792", "ITA", "0" },
    { "3", "0", "2N", "Ivano Fioravanti", "1788", "1788", "ITA", "0" },
    { "4", "0", "2N", "Filippo Melchiorre", "1675", "1675", "ITA", "0" },
    { "5", "0", "2N", "Andrea Fiore", "1619", "1619", "ITA", "0" },
    { "6", "0", "3N", "Massimo Bianchini", "1562", "1562", "ITA", "0" },
    { "7", "0", "NC", "Filippo Cavoli", "1448", "1448", "ITA", "0" },
    { "8", "0", "NC", "Cristian Spadoni", "1448", "1448", "ITA", "0" },
    { "9", "0", "NC", "Emanuele Marini", "1448", "1448", "ITA", "0" },
    { "10", "0", "NC", "Alejandro Gomez", "1448", "1448", "ITA", "0" },
    { "11", "0", "NC", "Lorenzo Caputi", "1391", "1391", "ITA", "0" },
    { "12", "0", "3N", "Mario Giovanzanti", "1368", "1368", "ITA", "0" },
    { "13", "0", "NC", "Stefano Cavoli", "1337", "1337", "ITA", "0" },
    { "14", "0", "NC", "Raffaele Gentile", "1311", "1311", "ITA", "0" },
    { "15", "0", "NC", "Erminio Castaldi", "1212", "1212", "ITA", "0" },
    { "16", "0", "NC", "Filippo Sciarra", "999", "999", "ITA", "0" } };

    String[][] firstRoundScoreboard = { { "1", "Leonardo Petrilli", "Emanuele Marini", "-"},
    { "2", "Alejandro Gomez", "Francesco Giovanzanti", "-"},
    { "3", "Ivano Fioravanti", "Lorenzo Caputi", "-"},
    { "4", "Mario Giovanzanti", "Filippo Melchiorre", "-"}, { "5", "Andrea Fiore", "Stefano Cavoli", "-"},
    { "6", "Raffaele Gentile", "Massimo Bianchini", "-"},
    { "7", "Filippo Cavoli", "Erminio Castaldi", "-"},
    { "8", "Filippo Sciarra", "Cristian Spadoni", "-"} };

    String[][] firstRoundStandings = { { "1", "1.0", "NC", "Lorenzo Caputi", "1391", "2188", "ITA", "1788.0" },
    { "2", "1.0", "3N", "Mario Giovanzanti", "1368", "2875", "ITA", "1675.0" },
    { "3", "1.0", "NC", "Stefano Cavoli", "1337", "2819", "ITA", "1619.0" },
    { "4", "1.0", "1N", "Leonardo Petrilli", "1922", "1848", "ITA", "1448.0" },
    { "5", "1.0", "1N", "Francesco Giovanzanti", "1792", "1848", "ITA", "1448.0" },
    { "6", "1.0", "NC", "Erminio Castaldi", "1212", "1848", "ITA", "1448.0" },
    { "7", "1.0", "3N", "Massimo Bianchini", "1562", "1711", "ITA", "1311.0" },
    { "8", "1.0", "NC", "Cristian Spadoni", "1448", "1399", "ITA", "999.0" },
    { "9", "0.0", "NC", "Emanuele Marini", "1448", "1522", "ITA", "1922.0" },
    { "10", "0.0", "NC", "Alejandro Gomez", "1448", "1392", "ITA", "1792.0" },
    { "11", "0.0", "NC", "Raffaele Gentile", "1311", "1162", "ITA", "1562.0" },
    { "12", "0.0", "NC", "Filippo Sciarra", "999", "1048", "ITA", "1448.0" },
    { "13", "0.0", "2N", "Ivano Fioravanti", "1788", "991", "ITA", "1391.0" },
    { "14", "0.0", "2N", "Filippo Melchiorre", "1675", "968", "ITA", "1368.0" },
    { "15", "0.0", "2N", "Andrea Fiore", "1619", "937", "ITA", "1337.0" },
    { "16", "0.0", "NC", "Filippo Cavoli", "1448", "812", "ITA", "1212.0" } };

    String[][] secondRoundScoreboard = { { "1", "Lorenzo Caputi", "Francesco Giovanzanti", "-"},
    { "2", "Erminio Castaldi", "Mario Giovanzanti", "-"},
    { "3", "Stefano Cavoli", "Massimo Bianchini", "-"},
    { "4", "Cristian Spadoni", "Leonardo Petrilli", "-"},
    { "5", "Emanuele Marini", "Ivano Fioravanti", "-"},
    { "6", "Filippo Melchiorre", "Alejandro Gomez", "-"}, { "7", "Andrea Fiore", "Raffaele Gentile", "-"},
    { "8", "Filippo Cavoli", "Filippo Sciarra", "-"} };

    rcl.startTournament();
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < 8; j++) {
            assertTrue(f.getStandings().getStandings()[i + 1][j].equals(initialRoundStandings[i][j]));
        }
    }

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 4; j++) {
            assertTrue(f.getScoreboard().getBoard()[i + 1][j].equals(firstRoundScoreboard[i][j]));
        }
    }

    String[] results = { "1-0", "0-1", "0-1", "1-0", "0-1", "0-1", "0-1", "0-1" };
    rcl.uploadResults(results);

    for (int i = 0; i < 8; i++) {
        assertTrue(f.getScoreboard().getBoard()[i + 1][3].equals(results[i]));
    }

    rcl.updateStandings();
    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < 8; j++) {
            assertTrue(f.getStandings().getStandings()[i + 1][j].equals(firstRoundStandings[i][j]));
        }
    }

    rcl.publishNewRound();

    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 4; j++) {
            assertTrue(f.getScoreboard().getBoard()[i + 1][j].equals(secondRoundScoreboard[i][j]));
        }
    }
}
```

Oltre al test case riportato in figura è stato implementato anche il test case *testOddPlayersTournament()* che caratterizza il caso in cui i partecipanti al torneo siano in numero dispari. In questa situazione, ad ogni turno e secondo alcune regole, viene scelto il giocatore che non potrà giocare. Come detto in precedenza per una visione completa dei test si rimanda alla repo Github sopra linkata.

Appendice A

A.1 Che cos'è il punteggio Elo?

Il **punteggio Elo** serve per esprimere numericamente la forza di un giocatore di scacchi; il nome deriva dall'ideatore del sistema, l'ingegnere statunitense di origine ungherese, **Arpad Emrick Elo**.

Il punteggio Elo si basa sulla distribuzione Gaussiana delle probabilità dell'evento "partita". Elo suggerì di stimare il rendimento dei giocatori in base al risultato ottenuto nella partita, considerando anche il punteggio dell'avversario. Se un giocatore vince più partite di quanto ci si aspetti il suo punteggio sale, se invece ne vince di meno il punteggio scende. Il punteggio Elo di un giocatore è rappresentato da un numero che può cambiare a seconda del risultato delle partite classificate giocate. La differenza nelle valutazioni tra i due avversari in una partita serve per prevedere il risultato della partita stessa: ci si aspetta che due giocatori con lo stesso punteggio ottengano una vittoria nel 50% dei casi, mentre un giocatore il cui punteggio è 100 punti superiore a quello del suo avversario dovrebbe ottenere il 64% delle vittorie, che salgono al 76% se la differenza è di 200 punti Elo.

Dopo ogni partita, il giocatore vincente prende punti da quello perdente. La differenza tra le valutazioni del vincitore e del perdente determina il numero totale di punti guadagnati o persi dopo una partita. Se il giocatore con il punteggio più alto vince, al giocatore con il punteggio più basso verranno presi solo pochi punti. Tuttavia, se il giocatore con il punteggio inferiore ottiene una vittoria, saranno trasferiti molti punti di valutazione. Il giocatore con il punteggio più basso guadagnerà anche alcuni punti dal giocatore con il punteggio più alto in caso di pareggio. Ciò significa che questo sistema di valutazione si corregge automaticamente. I giocatori le cui valutazioni sono troppo basse o troppo alte dovrebbero, a lungo termine, fare rispettivamente meglio o peggio di quanto previsto dal sistema di valutazione e quindi guadagnare o perdere punti di valutazione finché le valutazioni non riflettono la loro vera forza di gioco.

A.2 Swiss Pairing

Il sistema svizzero consiste nell'accoppiare turno per turno giocatori che hanno lo stesso punteggio in classifica oppure, se non può essere lo stesso, il punteggio più vicino.

L'accoppiamento del primo turno si realizza in base al punteggio Elo. Per solito i partecipanti si dividono in due gruppi: nel primo la metà con Elo più alto, nel secondo gli altri. Il miglior giocatore del primo lo si accoppia con il miglior giocatore del secondo, e così via. È abbastanza ovvio che, via via che i turni procedono, aumentano le possibilità che si incontrino giocatori di forza più o meno equivalente. In caso di pari punti nella classifica finale, si fa ricorso a sistemi di spareggio, i più noti dei quali sono il "Sonneborn-Berger" e il "Buchholz". Il secondo consiste nel sommare tutti i punti degli avversari incontrati, il primo somma invece tutti i punti degli avversari battuti, ma solo la metà dei punti degli avversari con i quali si è pattato. È tuttavia inevitabile che il fattore fortuna giochi un suo ruolo negli accoppiamenti e quindi nella classifica finale, specie quando si allarga la forbice fra il numero dei partecipanti e il numero dei turni. Non di rado infatti può accadere che i punti evidenziati dal Sonneborn-Berger o dal Buchholz del vincitore siano inferiori ai punti realizzati da chi giunge più in basso nella classifica finale.

A.3 ARO : Average Rating of Opponents

ARO è l'acronimo inglese di Average Rating of Opponents (rating medio degli avversari incontrati).

Fornisce un'indicazione della forza presunta degli avversari incontrati. Ha come vantaggio che raramente assume lo stesso valore per due o più giocatori a pari punti e come limite che prima di disputare l'ultima partita di torneo già se ne conosce il valore. Per tale motivo si preferisce usarlo come criterio secondario di spareggio.

A.4 Metodo '400

La performance è un calcolo ipotetico di forza che scaturisce dalle partite di un singolo torneo. Molte organizzazioni scacchistiche utilizzano **l'algoritmo 400** per effettuare il calcolo della performance. Essa indica semplicemente come un giocatore si sia comportato in un determinato torneo di scacchi, senza conoscerne la sua forza intrinseca. L'algoritmo si basa sui seguenti calcoli. Si devono sommare:

- L'ELO dei giocatori battuti +400;
- L'ELO dei giocatori con cui si è perso -400;
- L'ELO dei giocatori con cui si è pattato;
- Alla fine bisogna dividere la somma effettuata per il numero di partite giocate.

Appendice B

B.1 Use Case Templates

Use-case field	Description
Use-Case Name :	Show Tournament Details
Level :	User Goal
Actor :	Person
Preconditions :	L'utente è nella pagina iniziale.
Use-case overview :	L'utente visualizza tutti i dettagli di un torneo a cui è interessato.
Normal flow :	<ol style="list-style-type: none">1. L'utente seleziona un torneo della lista (vedasi MockUp a pag.13)2. Il sistema mostra tutte le informazioni relative al torneo selezionato.

Use-case field	Description
Use-Case Name :	Delete Account
Level :	User Goal
Actor :	Chess Person
Preconditions :	L'utente ha eseguito correttamente il login.
Normal flow :	<ol style="list-style-type: none">1. L'utente richiede la cancellazione del suo account.2. Il sistema richiede all'utente di inserire la password del suo account.3. L'utente inserisce la password corretta.4. Il sistema cancella l'account.
Alternative flow :	<p>3.a L'utente inserisce una password sbagliata. 4.a Il sistema mostra un messaggio di errore invitando l'utente a riprovare.</p>

Appendice C

C.1 Publish New Round

```
public void publishNewRound(Standings standings) {  
    ArrayList<String[]> playersNames = new ArrayList<String[]>();  
    ArrayList<String[]> copyPlayersNames = new ArrayList<String[]>();  
    int numOfPairings = 0;  
  
    for (int i = 1; i < numOfPlayers + 1; i++) {  
        playersNames.add(standings.getElement(i));  
    }  
  
    boolean found = false;  
    String[] tmp;  
  
    if (playersNames.size() % 2 == 1) {  
        int m = playersNames.size() - 1;  
  
        while (found == false) {  
            if (!(previousPairings.get(playersNames.get(m)[3]).contains("_BYE_"))) {  
                found = true;  
                board[(int) Math.ceil((double)numOfPlayers / 2)][1] = playersNames.get(m)[3];  
                board[(int) Math.ceil((double)numOfPlayers / 2)][2] = "_BYE_";  
                board[(int) Math.ceil((double)numOfPlayers / 2)][3] = "_";  
                previousPairings.get(playersNames.get(m)[3]).add("_BYE_");  
                playersNames.remove(m);  
            }  
            m--;  
        }  
        found = false;  
    }  
  
    while (playersNames.size() != 0) {  
        do {  
            copyPlayersNames.add(playersNames.get(0));  
            tmp = playersNames.remove(0);  
        } while (playersNames.size() != 0 && tmp[1].equals(playersNames.get(0)[1]));  
  
        if (copyPlayersNames.size() % 2 == 1 && copyPlayersNames.size() != 1) {  
            playersNames.add(0, copyPlayersNames.remove(copyPlayersNames.size() - 1));  
        } else if (copyPlayersNames.size() == 1) {  
            int index = 0;  
            while (found == false) {  
                if (!(previousPairings.get(copyPlayersNames.get(0)[3]).contains(playersNames.get(index)[3]))) {  
                    found = true;  
                    previousPairings.get(copyPlayersNames.get(0)[3]).add(playersNames.get(index)[3]);  
                    previousPairings.get(playersNames.get(index)[3]).add(copyPlayersNames.get(0)[3]);  
                }  
                index++;  
            }  
        }  
    }  
}
```

```

numOfPairings++;
board[numOfPairings][3] = "-";
if (previousColour.get(copyPlayersNames.get(0)[3]).equals("WHITE")) {
    previousColour.put(copyPlayersNames.get(0)[3], "BLACK");
    previousColour.put(playersNames.get(index)[3], "WHITE");
    board[numOfPairings][1] = playersNames.get(index)[3];
    board[numOfPairings][2] = copyPlayersNames.get(0)[3];
} else {
    previousColour.put(copyPlayersNames.get(0)[3], "WHITE");
    previousColour.put(playersNames.get(index)[3], "BLACK");
    board[numOfPairings][2] = playersNames.get(index)[3];
    board[numOfPairings][1] = copyPlayersNames.get(0)[3];
}

copyPlayersNames.remove(0);
playersNames.remove(index);

}

index++;

}

found = false;

}

while (copyPlayersNames.size() != 0) {

    int k = (int) Math.ceil(((double)copyPlayersNames.size() / 2));

    if (copyPlayersNames.size() != 1) {

        do {
            if (!(previousPairings.get(copyPlayersNames.get(0)[3]).contains(copyPlayersNames.get(k)[3])) || playersNames
                found = true;
                previousPairings.get(copyPlayersNames.get(0)[3]).add(copyPlayersNames.get(k)[3]);
                previousPairings.get(copyPlayersNames.get(k)[3]).add(copyPlayersNames.get(0)[3]);
                numOfPairings++;
                board[numOfPairings][3] = "-";

                if (previousColour.get(copyPlayersNames.get(0)[3]).equals("WHITE")) {

                    previousColour.put(copyPlayersNames.get(0)[3], "BLACK");
                    previousColour.put(copyPlayersNames.get(k)[3], "WHITE");
                    board[numOfPairings][1] = copyPlayersNames.get(k)[3];
                    board[numOfPairings][2] = copyPlayersNames.get(0)[3];
                } else {
                    previousColour.put(copyPlayersNames.get(0)[3], "WHITE");
                    previousColour.put(copyPlayersNames.get(k)[3], "BLACK");
                    board[numOfPairings][1] = copyPlayersNames.get(0)[3];
                }
            } while (true);
        } while (true);
    }
}

```

```

        board[numOfPairings][2] = copyPlayersNames.get(k)[3];
    }

    copyPlayersNames.remove(k);
    copyPlayersNames.remove(0);

}

if (k == copyPlayersNames.size() - 1) {
    k = 1;
} else {
    k++;
}

} while (found == false && k != (int) Math.ceil((double)copyPlayersNames.size() / 2));
}

if (found == false) {
    int l = 0;

    while (found == false) {
        if (!(previousPairings.get(copyPlayersNames.get(0)[3]).contains(playersNames.get(l)[3]))) {
            found = true;
            previousPairings.get(copyPlayersNames.get(0)[3]).add(playersNames.get(l)[3]);
            previousPairings.get(playersNames.get(l)[3]).add(copyPlayersNames.get(0)[3]);
            numOfPairings++;
            board[numOfPairings][3] = "-";

            if (previousColour.get(copyPlayersNames.get(0)[3]).equals("WHITE")) {
                previousColour.put(copyPlayersNames.get(0)[3], "BLACK");
                previousColour.put(playersNames.get(l)[3], "WHITE");
                board[numOfPairings][1] = playersNames.get(l)[3];
                board[numOfPairings][2] = copyPlayersNames.get(0)[3];
            } else {
                previousColour.put(copyPlayersNames.get(0)[3], "WHITE");
                previousColour.put(playersNames.get(l)[3], "BLACK");
                board[numOfPairings][1] = copyPlayersNames.get(0)[3];
                board[numOfPairings][2] = playersNames.get(l)[3];
            }

            copyPlayersNames.remove(0);
            playersNames.remove(l);
        }
        l++;
    }
}

}

found = false;
}

}

}

```