

# **Jhaturanga**

## Relazione di progetto

## Programmazione ad Oggetti

Andruccioli Manuel  
Mazzoli Alessandro  
Patriti Tommaso  
Scolari Stefano

18 aprile 2021

# Indice

<b>1 Analisi</b>	<b>3</b>
1.1 Requisiti . . . . .	4
1.2 Analisi e modello del dominio . . . . .	5
<b>2 Design</b>	<b>7</b>
2.1 Architettura . . . . .	7
2.1.1 Model . . . . .	8
2.1.2 View . . . . .	9
2.1.3 Controller . . . . .	10
2.2 Design dettagliato . . . . .	12
2.2.1 Andruccioli Manuel . . . . .	12
2.2.2 Mazzoli Alessandro . . . . .	17
2.2.3 Tommaso Patriti . . . . .	29
2.2.4 Scolari Stefano . . . . .	35
<b>3 Sviluppo</b>	<b>44</b>
3.1 Testing automatizzato . . . . .	44
3.1.1 Andruccioli Manuel . . . . .	44
3.1.2 Scolari Stefano, Mazzoli Alessandro . . . . .	45
3.1.3 Patriti Tommaso . . . . .	46
3.2 Metodologia di lavoro . . . . .	46
3.2.1 Andruccioli Manuel . . . . .	47
3.2.2 Mazzoli Alessandro . . . . .	47
3.2.3 Patriti Tommaso . . . . .	48
3.2.4 Scolari Stefano . . . . .	48
3.3 Note di sviluppo . . . . .	48
3.3.1 Andruccioli Manuel . . . . .	48
3.3.2 Mazzoli Alessandro . . . . .	48
3.3.3 Patriti Tommaso . . . . .	49
3.3.4 Scolari Stefano . . . . .	50
3.3.5 Crediti . . . . .	50

<b>4 Commenti finali</b>	<b>51</b>
4.1 Autovalutazione e lavori futuri . . . . .	51
4.1.1 Andruccioli Manuel . . . . .	51
4.1.2 Mazzoli Alessandro . . . . .	51
4.1.3 Patriti Tommaso . . . . .	52
4.1.4 Scolari Stefano . . . . .	53
4.2 Difficoltà incontrate e commenti per i docenti . . . . .	53
4.2.1 Andruccioli Manuel . . . . .	53
4.2.2 Mazzoli Alessandro . . . . .	53
4.2.3 Patriti Tommaso . . . . .	54
4.2.4 Scolari Stefano . . . . .	54
<b>A Guida utente</b>	<b>56</b>
A.1 Selezione Piattaforma . . . . .	56
A.2 Login . . . . .	57
A.3 Home Page . . . . .	58
A.4 Settings . . . . .	58
A.5 Leaderboard . . . . .	59
A.6 History . . . . .	60
A.7 Nuova Partita . . . . .	61
A.8 Modalità Online . . . . .	62
A.9 Modalità Offline . . . . .	62
A.10 Selezione Modalità . . . . .	63
A.11 Interfaccia Di Gioco . . . . .	64
A.12 Scacchiera Personalizzata . . . . .	65
A.13 Problemi Scacchistici . . . . .	65
A.14 Resource Pack . . . . .	65
<b>B Esercitazioni di laboratorio</b>	<b>67</b>
B.1 Andruccioli Manuel . . . . .	67
B.2 Mazzoli Alessandro . . . . .	67
B.3 Patriti Tommaso . . . . .	68
B.4 Scolari Stefano . . . . .	68

# Capitolo 1

## Analisi

Il Team si pone come obiettivo lo sviluppo di un applicativo per giocare agli scacchi e molte sue varianti.

Il titolo dell'applicativo deriva dal nome del gioco originale, evolutosi poi negli scacchi, che si chiamava "Chaturanga".

Gli scacchi sono un gioco di strategia che si svolge su una tavola quadrata detta scacchiera, formata da 64 caselle di due colori alternati, sulla quale ogni giocatore dispone di 16 pezzi (bianchi o neri: "il Bianco" e "il Nero" designano i due sfidanti). Ogni giocatore dispone di:

- Un re.
- Una donna (o "regina").
- Due alfieri.
- Due cavalli.
- Due torri.
- Otto pedoni.

Ogni casella può essere occupata da un solo pezzo, caratterizzato dal proprio movimento, che può catturare o "mangiare" il pezzo avversario andando a occuparne la casella; obiettivo del gioco è dare scacco matto, ovvero minacciare la cattura del re avversario, in modo tale che l'altro giocatore venga messo nell'impossibilità di evitarla con mosse legali.

Inoltre l'applicativo permette di risolvere problemi scacchistici, ovvero: data una scacchiera con i pezzi già distribuiti con una certa disposizione (come se la partita fosse già iniziata), il giocatore deve essere in grado di scegliere ad ogni mossa quella che il problema ritiene essere la migliore.

Tra le modalità di gioco, sono inoltre disponibili alcune varianti agli scacchi classici, detti scacchi eterodossi, basate sugli scacchi, ma con un certo numero di cambiamenti. Questi possono includere:

- Posizione iniziale differente
- Differenti obiettivi (farsi dare scacco matto, perdere tutti i pezzi)
- Regole differenti per movimento, cattura e scacco

## 1.1 Requisiti

### Requisiti funzionali

- L'applicazione permette di eseguire i movimenti e mosse dei vari pezzi.
- L'applicazione dovrà essere in grado di gestire una classica partita di scacchi, applicando quindi le regole di base di quest'ultima.
- L'utente avrà la possibilità di scegliere il tema dell'applicazione, e dei pezzi.
- L'applicazione permetterà la scelta della modalità di gioco.
- Le partite potranno essere a tempo, ovvero, ad ogni giocatore viene associato un lasso temporale rappresentato da un timer che verrà messo in funzione ogni volta che sarà il turno di quel giocatore, e andrà in pausa azionando quello dell'avversario una volta fatta la mossa, per poi riazionarsi quando l'avversario avrà fatto la sua mossa. Quando il timer di un giocatore raggiunge lo zero, quest'ultimo perde automaticamente la partita pur non essendo in posizione di scacco matto.
- Il giocatore deve avere la possibilità di scegliere di ritirarsi dalla partita.
- Durante le partite sarà possibile navigare temporalmente tra le mosse eseguite precedentemente per permettere al giocatore di visualizzare in maniera più approfondita l'andamento dell'incontro.
- L'applicazione potrà gestire più profili utente mediante un login con password in modo da conservare i dati in maniera permanente tra una sessione di gioco e l'altra. Vi sarà anche la possibilità di giocare come GUEST.
- Sarà possibile visualizzare una classifica dei vari utenti che hanno giocato.
- Al termine della partita questa verrà salvata con la possibilità quindi di visualizzarne il replay in un secondo momento.
- Saranno evidenziate le celle dove è possibile muoversi.
- Sarà possibile giocare attraverso la rete contro un altro giocatore.
- L'applicazione permetterà di giocare a diverse varianti scacchistiche come opzione agli scacchi base.

- Sarà possibile accedere ad una modalità per la risoluzione di problemi scacchistici.
- Gestione dei suoni.

### **Requisiti non funzionali**

- L'applicazione dovrà garantire una sicurezza minima nel salvataggio delle password di ogni utente tramite funzioni di hashing.
- La serializzazione delle partite per il relativo salvataggio su file e il successivo caricamento dovranno essere effettuate in modo efficiente riducendo al minimo le informazioni che andranno salvate.
- L'organizzazione dei file di configurazione e delle texture dovrà essere ottimizzata tramite una strutturazione standard in modo da velocizzare le ricerche.

## **1.2 Analisi e modello del dominio**

Jhaturanga dovrà essere in grado di gestire partite di scacchi, sia di tipo classico sia di diverse varianti.

Il tipo di partita è definito dal GameType, il quale avrà le sue specifiche regole concernenti i movimenti dei singoli pezzi (Piece), il numero e tipo di questi ultimi e le condizioni di vittoria o pareggio. Sarà quindi fondamentale progettare l'applicativo in maniera tale da rendere semplice la gestione ed implementazione delle varianti.

Una partita, detta Match, conterrà le principali informazioni riguardo l'incontro corrente:

- I giocatori, identificati dall'entità Player ciascuno.
- La condizione attuale della scacchiera, rappresentata dall'entità Board.
- Il Timer, utilizzato per dettare i tempi di gioco rimasti ad ogni Player.
- Il Player vincitore, al momento della conclusione della partita.

La Board contiene le informazioni riguardo la posizione dei pezzi.

I pezzi prendono il nome di Piece, ognuno dei quali ha specifiche regole riguardo i propri movimenti sulla Board, un nome, l'attuale posizione ed il Player che lo manovra.

I movimenti (Movement), contengono le informazioni riguardo una mossa, ovvero le posizioni (Position) di inizio e fine per ogni pezzo (Piece) interessato.

Una delle principali difficoltà sarà quella di riuscire a gestire la diversità dei GameType, in modo da rendere l'architettura ampiamente estendibile ad un numero molto elevato di varianti.

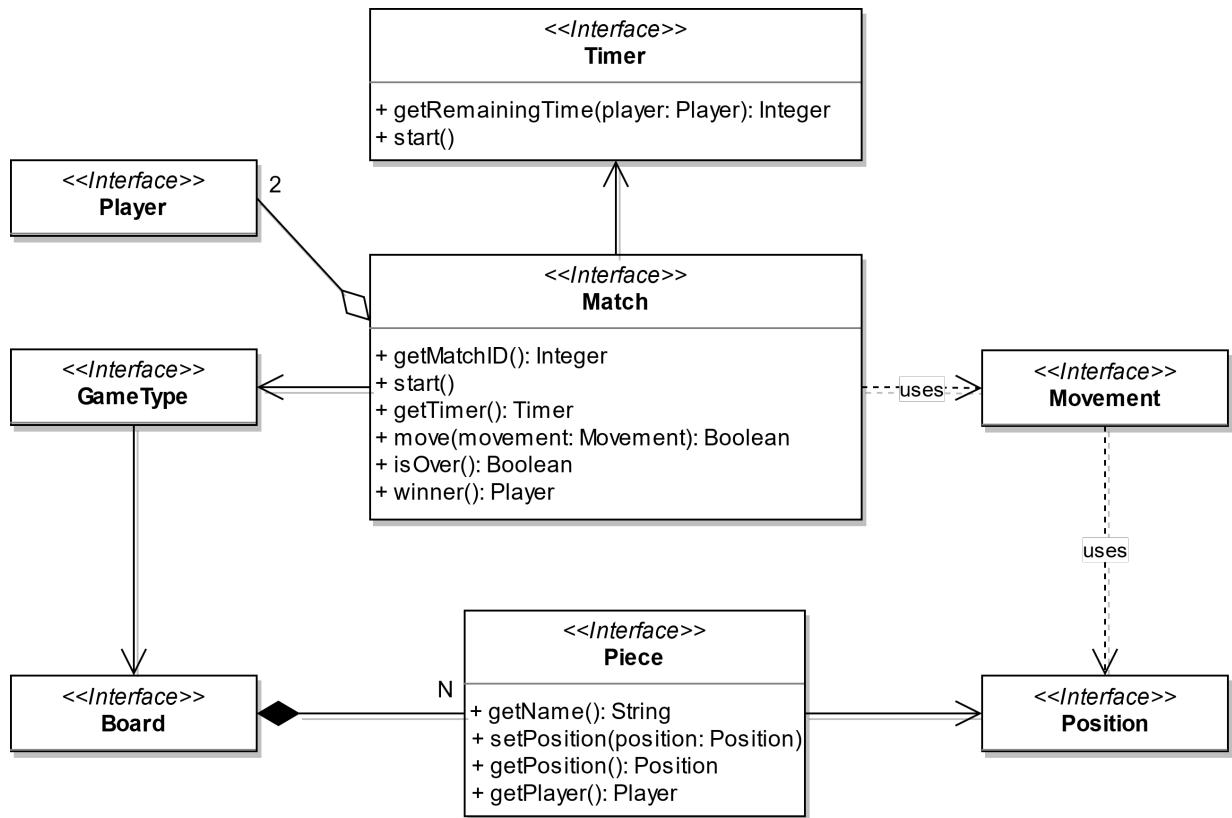


Figura 1.1: Schema UML dell’analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

# Capitolo 2

## Design

### 2.1 Architettura

Per lo sviluppo dell'architettura di Jhaturanga si è deciso di seguire il pattern architettonico MVC (Model - View - Controller).

A prova dell'estendibilità dell'applicativo dal punto di vista di MVC, abbiamo deciso di implementare anche una versione proof-of-concept da Command Line, che integra una versione basilare dell'intera applicazione seppur completamente funzionante.

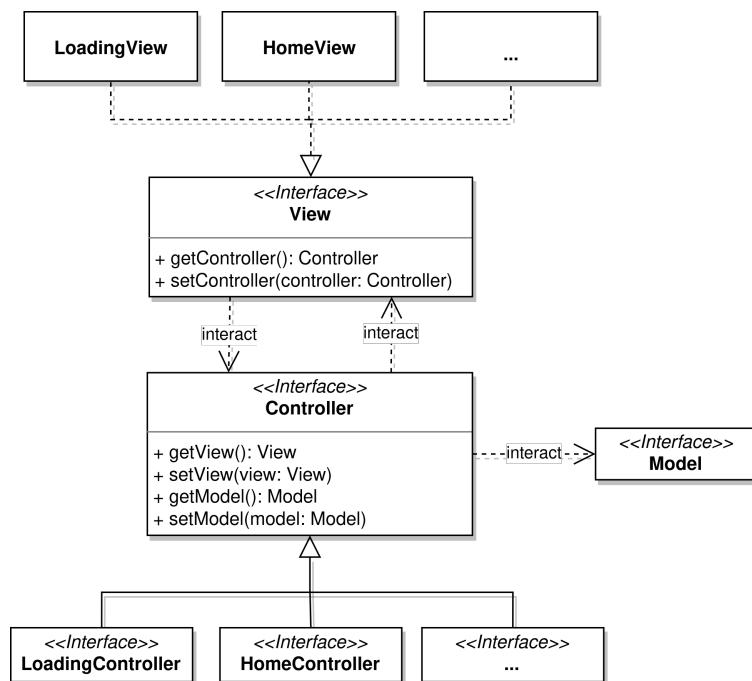


Figura 2.1: Architettura MVC da noi implementata

### 2.1.1 Model

Il Model si occupa di gestire le metodologie di accesso e modifica dei dati dell'applicazione e del dominio applicativo, determinando coerentemente come le interazioni dell'utente influenzino lo stato corrente dell'applicazione.

Scopo del model sarà quindi fornire al Controller un accesso allo stato attuale dell'applicazione.

Per fornire tale funzionalità è stato scelto di utilizzare un'entità **Model**, responsabile di tenere al suo interno lo stato attuale dell'applicazione, concernente quindi i principali componenti come gli utenti e il match a cui si sta giocando.

L'effettiva implementazione consiste poi nella classe **ApplicationInstance**.

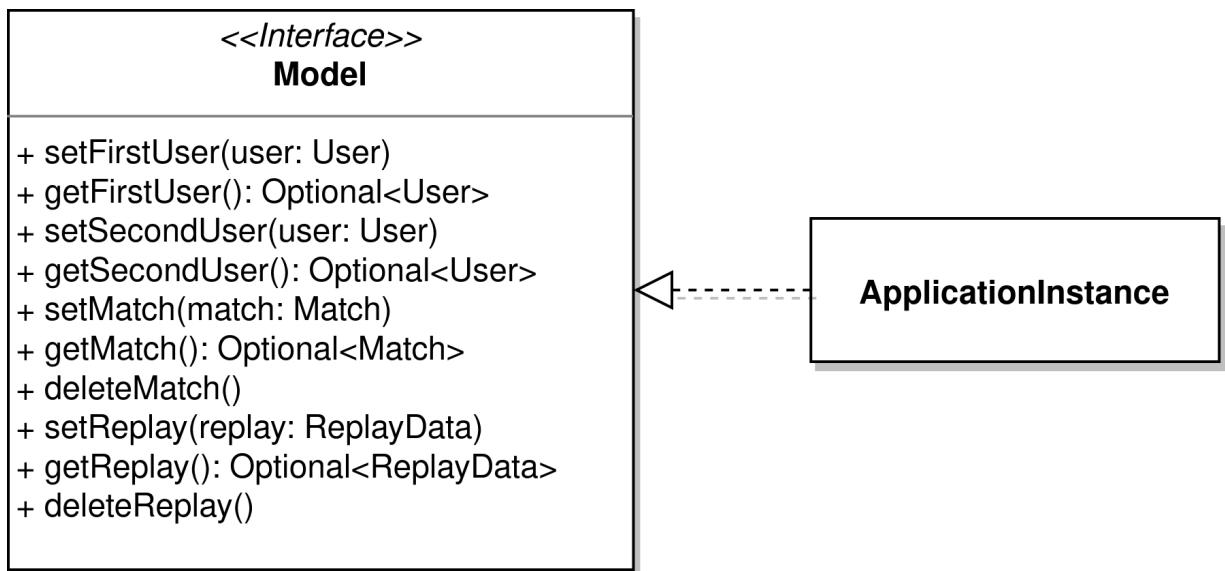


Figura 2.2: Schema esemplificativo del Model

## 2.1.2 View

La **View** consiste nella schermata dell'applicazione che si occupa della gestione della parte grafica, della User Experience e dell'interazione con l'utente.

È compito delle varie **View** registrare ed informare il rispettivo **Controller** di interazioni con l'applicazione da parte dell'utente, in attesa di una sua risposta sul cambiamento dei dati.

Nonostante sia stato scelto sin da subito di utilizzare JavaFx, fin dalla prima fase di analisi si è deciso di strutturare la parte di **View** e **Controller** in modo indipendente dal framework utilizzato.

Abbiamo quindi fatto in modo che l'entità **Controller** non sia collegata all'implementazione delle **View** come invece viene fatto quando si utilizza JavaFX che vuole un "Controller" attaccato al file fxml. Avendo quindi isolato i **Controller** dal framework utilizzato abbiamo fatto in modo che cambiare libreria grafica implichi solamente la riscrittura della parte di **View**, lasciando invariata la parte di **Controller** e **Model**.

Poichè l'applicazione è stata strutturata per avere un numero abbastanza elevato di pagine, ognuna delle quali operante su dati e parti del model differenti, si è deciso di utilizzare un' associazione 1:1 tra **View** e **Controller**. Ovvero: ogni **View** ha associata a sè il relativo **Controller**, quest' ultimo responsabile di fornire e lavorare solamente sui dati richiesti dal contesto di quella pagina.

È proprio grazie a quest' architettura che ci è risultato estremamente semplice e triviale sviluppare la versione da Command-Line.

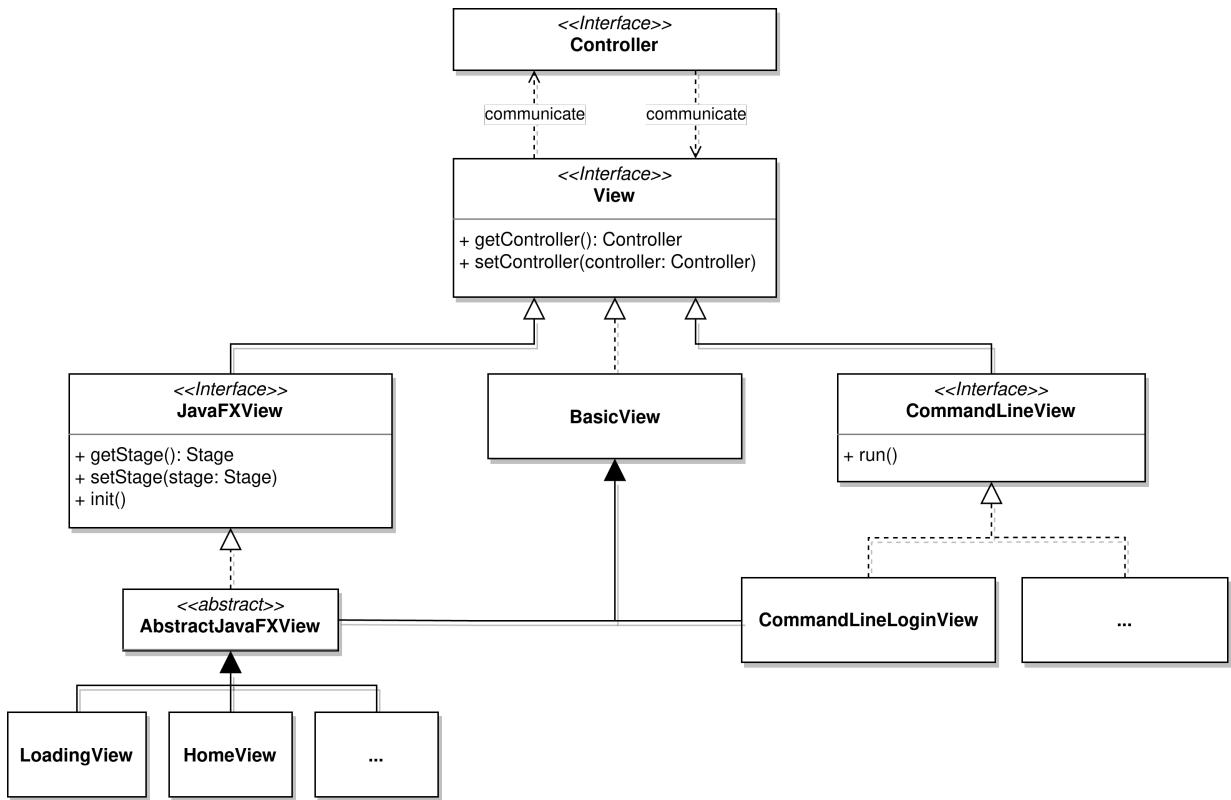


Figura 2.3: Schema UML dell’architettura MVC da noi implementata

### 2.1.3 Controller

Il **Controller** è quella componente cui spetta gestire in maniera conseguenziale le interazioni da parte dell’utente nella **View**, comunicando quindi al **Model** il cambiamento avvenuto.

Una volta che il **Model** avrà completato l’elaborazione della richiesta di cambiamento, il **Controller** avviserà la propria **View**, in modo tale che quest’ultima possa aggiornarsi in maniera coerente secondo le regole specificate dal **Model**.

Come già accennato nella sezione precedente, si è scelto di evitare l’utilizzo di un singolo controller, sia per i motivi sopracitati, sia perché un unico controller sarebbe risultato essere una "God Class", avente un eccessivo grado di responsabilità.

Abbiamo deciso quindi di organizzare anche il lato controller secondo una struttura gerarchica, dove l’entità del **Controller** è rappresentata dall’interfaccia che ne incapsula il funzionamento, ossia avere l’accesso al **Model** e alla **View**.

Da questo derivano poi le varie specializzazioni dei vari **Controller**, ognuno specifico secondo il contratto delle funzionalità, determinanti queste quali parti del **Model** concernono gli accessi del suddetto.

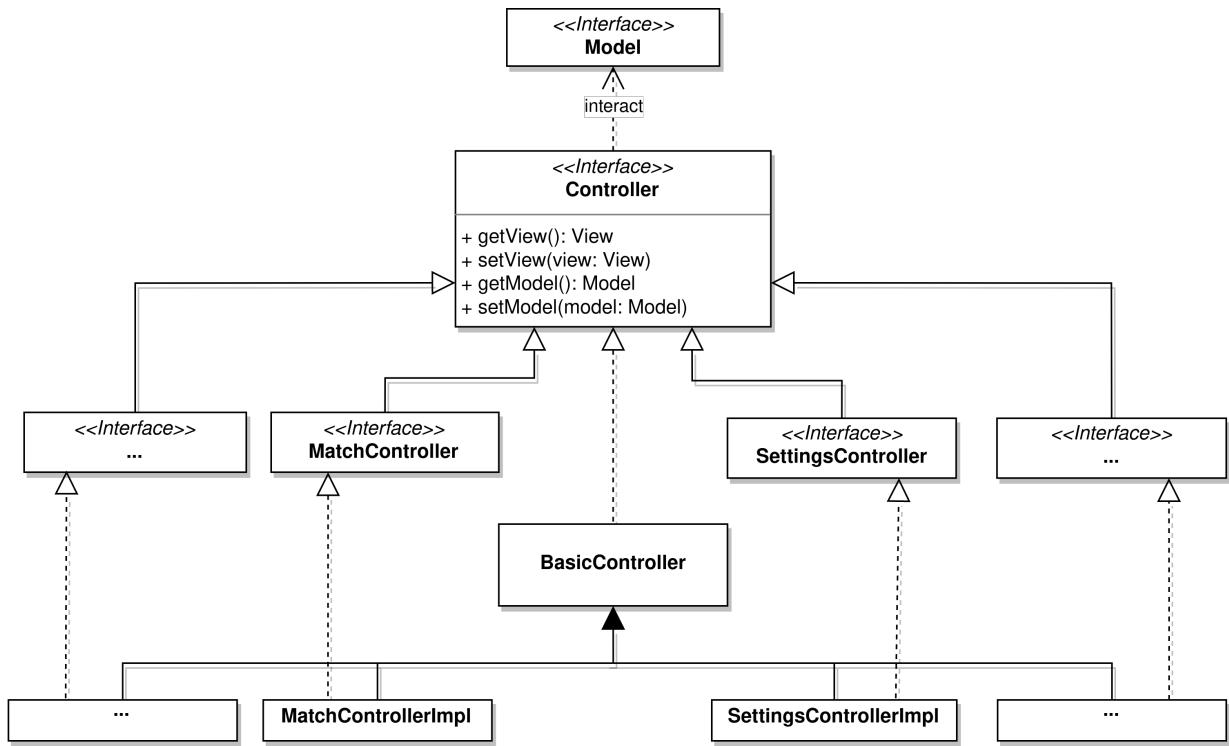


Figura 2.4: Organizzazione dei Controller nell'architettura MVC

Quello che ne è derivato è stata un architettura dell'applicazione con un alto livello di estendibilità, che ha agevolato di molto lo sviluppo, poiché ogni pagina e rispettivo **Controller** potevano lavorare indipendentemente dalle altre coppe **View-Controller**, rendendo possibile, e soprattutto molto semplice, integrare nuove funzionalità.

## 2.2 Design dettagliato

### 2.2.1 Andruccioli Manuel

Fin dalla fase di analisi, miravamo ad avere un'applicazione con la possibilità di mantenere i dati relativi ai risultati delle partite dei giocatori, per poi poterli visualizzare successivamente.

La parte della quale mi sono occupato maggiormente è stata la realizzazione di un sotto sistema per la memorizzazione persistente dei dati, il recupero di essi e la possibilità di riorganizzarli attraverso un classifica.

#### User

A fronte di un'accurata analisi, si è deciso di separare l'entità User da Player, trattando quest'ultima in una sezione apposita.

Lo scopo principale di **User** è quello di encapsulare l'utente che si autentica, per poi tener traccia dei risultati delle partite (vittoria, sconfitta o pareggio).

Per le funzionalità basilari che dobbiamo offrire, ho deciso di rendere univoco lo username dell'utente, che verrà riservato durante la registrazione e, in coppia con la password inserita, serviranno per potersi autenticare successivamente.

Per la creazione degli Users ho deciso di adottare il pattern **Builder**, per avere la possibilità di spezzare le fasi di creazione se necessario. Attualmente User contiene poche informazioni, ma se in futuro si vorranno aggiungere responsabilità, questa entità, così facendo, sarà facilmente scalabile. Inoltre ho pensato che questa procedura di creazione potrebbe tornare utile nel caso in cui si voglia generare giocatori fintizi, con dati random, nelle partite contro il computer.

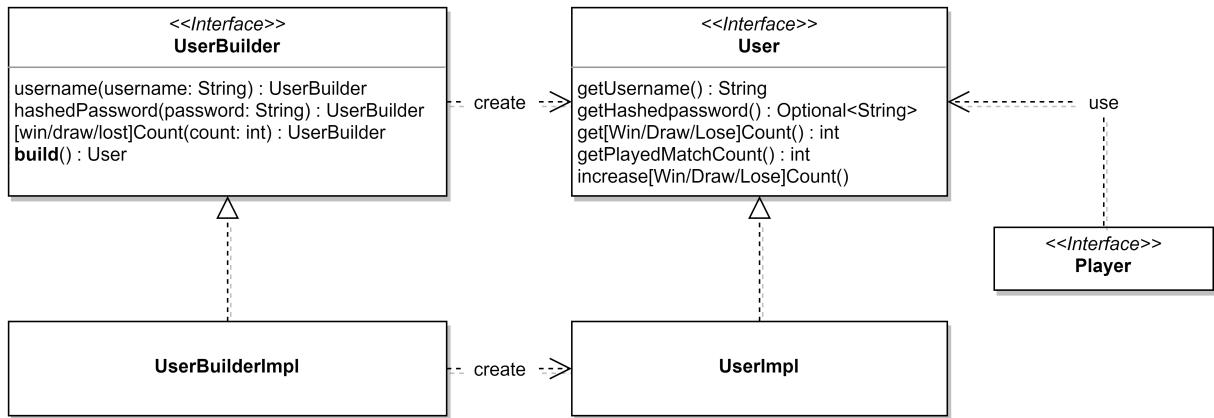


Figura 2.5: User e creazione

## Function Concatenator

Dovendo richiedere dati all'utente per inserire username e password, ho ideato questa interfaccia per facilitare ed estendere, il controllo su di essi ed ottenere un determinato risultato al termine dei test proposti.

L'intento principale è quello di avere la possibilità di concatenare tra loro funzioni dello stesso tipo, per poi generarne una nuova. Entrambe le operazioni non sono determinate, demandando il comportamento proprio alle implementazioni, realizzando ciò di cui si ha necessità.

Infatti, ho realizzato `StringValidatorImpl` che si occupa di concatenare test (funzioni che accettano una Stringa e restituiscono un risultato), per poi restituire una nuova funzione, costruita sulla base di ciò che è stato fornito. Come risultato ho voluto utilizzare una `enum`, aggiungendo un messaggio esplicativo.

Per facilitare la creazione, ho disposto una classe che contiene alcune funzioni di test già implementate.

Una volta creata la funzione validatrice, basterà applicarla ad una stringa, ottenendo come risultato 'CORRECT' se passa tutti i test, oppure il primo risultato di errore.

Questo approccio utilizzato, secondo me, è facilmente estendibile e/o modificabile, andando ad agire o creando nuovi test.

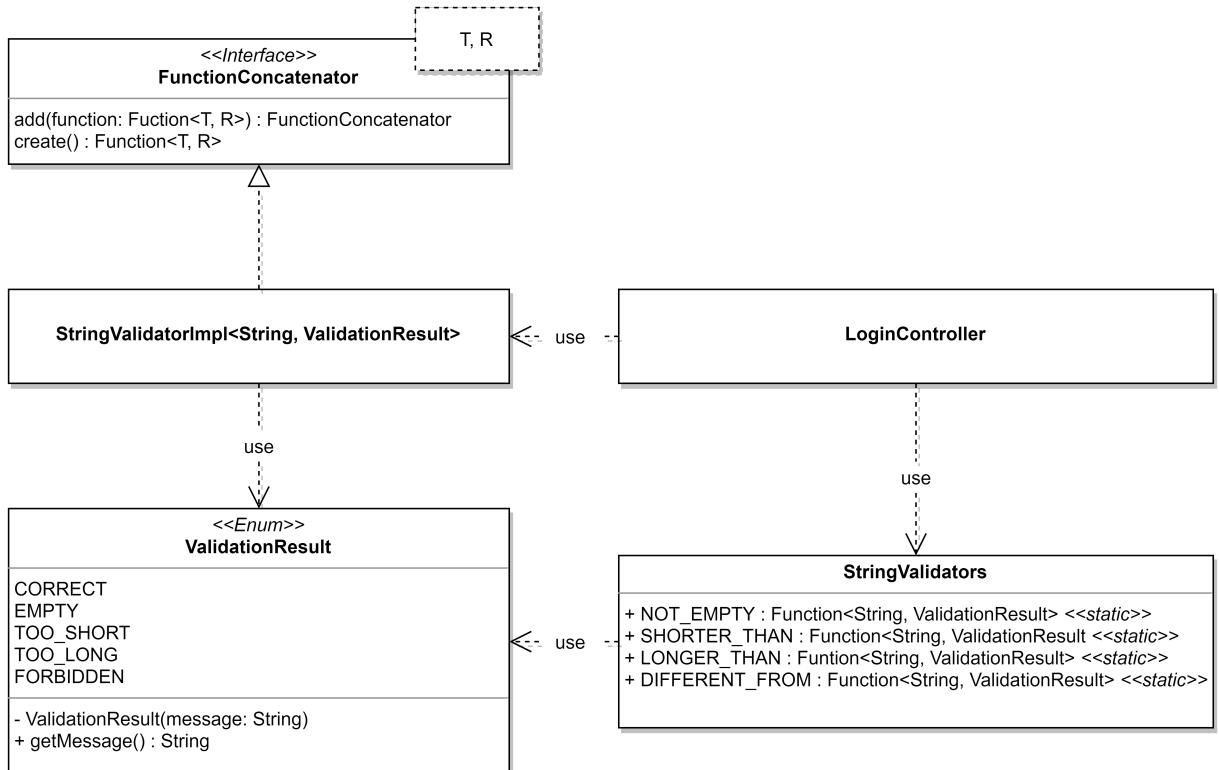


Figura 2.6: Concatenatore di funzioni

## User Manager

Nel momento in cui è giunta la necessità di salvare i dati degli utenti in una memoria persistente, ho realizzato fosse corretto separare le responsabilità delle varie parti in gioco, per poter riutilizzare il codice.

Da questa premessa infatti, ho realizzato lo **UserManager**, un’interfaccia che stabilisce le varie operazioni fattibili sulla collezione di utenti dell’applicazione. Avrei potuto implementare qua dentro direttamente il salvataggio e recupero dei dati su file, ma non sarebbe stato molto estendibile.

Così facendo, ho deciso di adottare il pattern **Strategy**: l’implementazione riceve una strategia mediante il costruttore, per salvare i dati, della quale si compone e delega ad essa le varie operazioni di I/O.

La **UsersDataStorageStrategy** difatti, sancisce le operazioni eseguibili sui dati in memoria persistente, implementando gli opportuni meccanismi, ad esempio, per andare a leggere e scrivere su file. Questo grado di libertà infatti, permette di modificare facilmente il metodo di archiviazione, scrivendo una nuova strategia. Ad esempio è possibile scrivere su file di testo, utilizzare uno stream di byte, ma anche impiegare un database. Io ho realizzato una semplice **JsonStrategy**, che sfrutta per l’appunto il formato JSON.

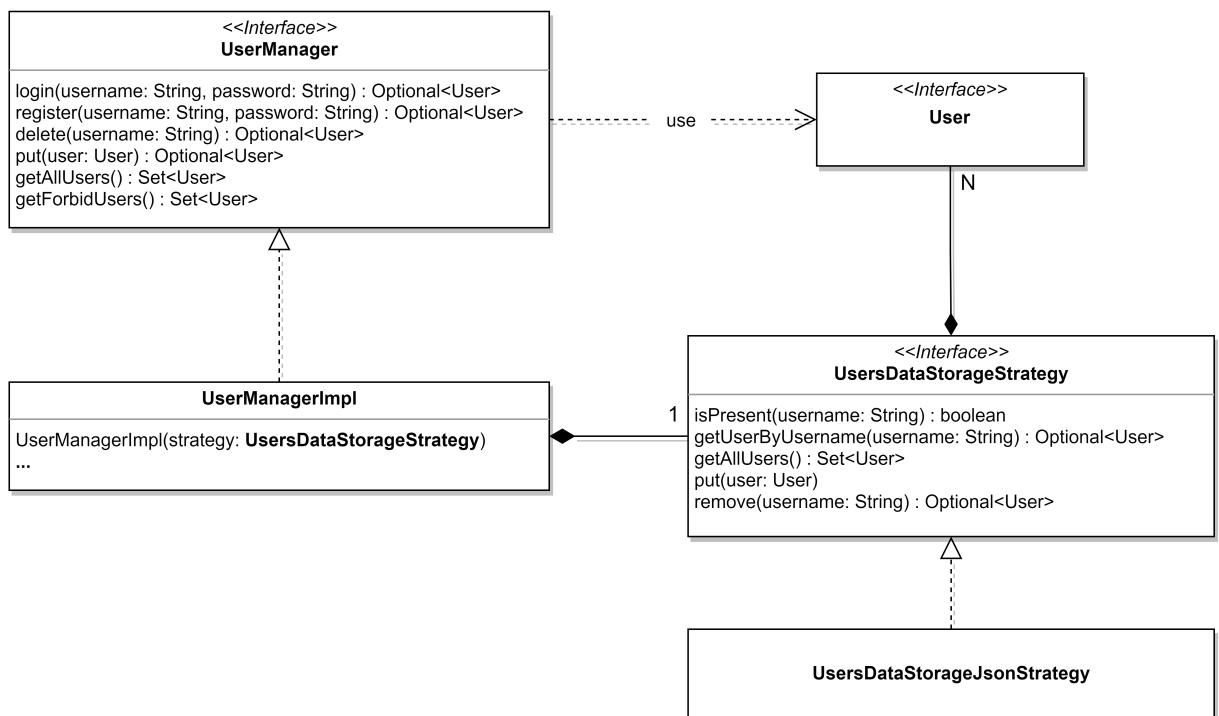


Figura 2.7: User Manager per gli utenti dell’applicazione

## Accesso unificato allo User Manager

Dopo la realizzazione dello **User Manager** si sarebbe potuto verificare un inconveniente non da poco. Nel caso in cui si scrivessero più strategie, non comunicanti tra di loro (e.g. salvano gli utenti su due file diversi), si avrebbe una frammentazione dei dati, che comprometterebbe la veridicità degli stessi.

Proprio per questo motivo, ho deciso di adottare il pattern **Singleton**, per rendere unico nell'applicazione, l'accesso allo User Manager. Così facendo posso ovviare ai problemi riportati sopra, senza però limitare la struttura precedentemente realizzata. Infatti, nel caso in cui si voglia cambiare la strategia di salvataggio, basterà modificare all'interno di `UserManagerSingleton`.

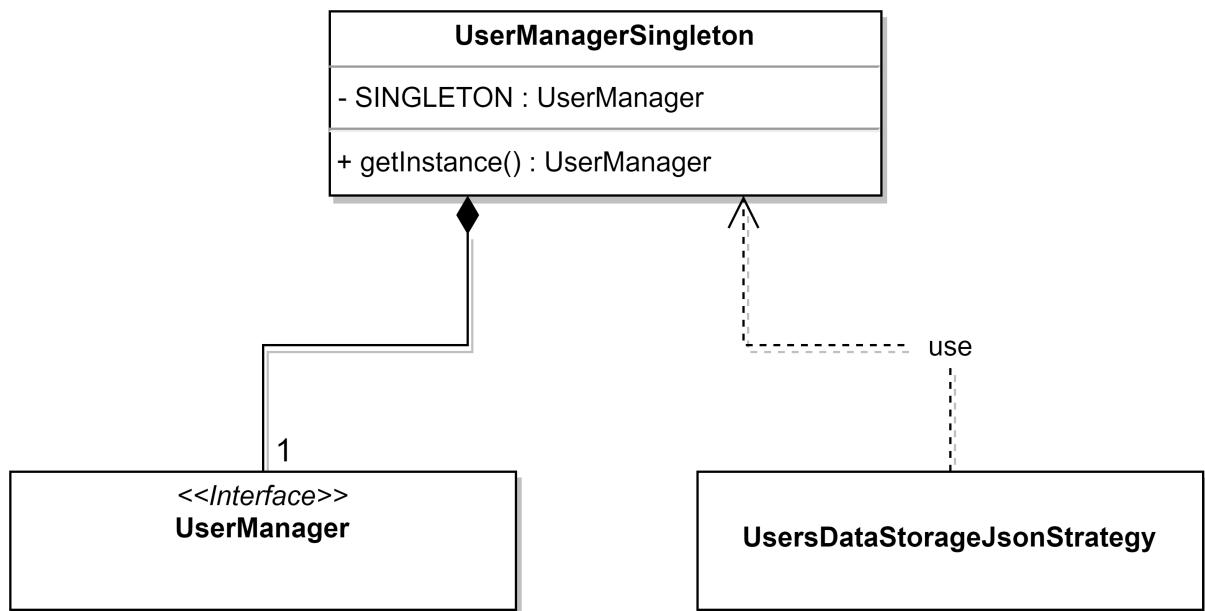


Figura 2.8: Singleton utilizzato per un accesso unificato

## Classifica

Nel momento in cui ho dovuto iniziare a pensare alla classifica, sarebbe bastato prendere tutti gli utenti dell'applicazione, con annessi i loro punteggi, e stamparli ove necessario. In questo caso non avrei dovuto implementare nient'altro, sarebbe bastato lo **User Manager**. Questa soluzione avrebbe soddisfatto i requisiti minimi di ciò che avevamo deciso, ma non sarebbe stata efficiente, infatti ogni funzionalità aggiuntiva avrebbe comportato ulteriore codice.

Fatta questa premessa infatti, ho pensato che fosse giusto l'utilizzo del pattern **Builder**, il quale mi avrebbe lasciato libertà di dividere la creazione della **Leaderboard**. Se in futuro aumentassero le responsabilità della classifica, non si andrebbe ad intaccare il codice già

scritto. Con i metodi `addUsers()`, `addFilter()` e `comparator()` è possibile manipolare i dati, creando classifiche parziali e non, mentre su `strategy()` ci torneremo in seguito. Sulla definizione dell’interfaccia `Leaderboard` ho notato non fosse corretto utilizzare una lista di `User`, proprio perché in questa situazione non sono necessarie tutte le informazioni interne dello stesso (e.g. password). Questo fatto mi ha portato ad utilizzare il pattern **Adapter**: la `LeaderboardUserAdapter` è l’interfaccia che cattura i dati dell’utente che realmente andranno in classifica. Infatti, questo pattern mi permette di aggiungere metodi specifici, ove necessari.

Proprio sul fatto di aggiungere responsabilità, ho deciso di assegnare un punteggio all’utente da andare ad inserire nella classifica: questa modifica non va ad intaccare lo `User`, grazie alle scelte illustrate in precedenza.

Per la gestione del punteggio ho pensato di utilizzare il pattern **Strategy**: questa scelta mi lascia la libertà di attribuire punteggi in base alle necessità.

Ho deciso di passare la strategia al Builder, proprio per avere congruenza sui dati: questo mi permette infatti di avere la stessa valutazione di punteggio all’interno della medesima classifica.

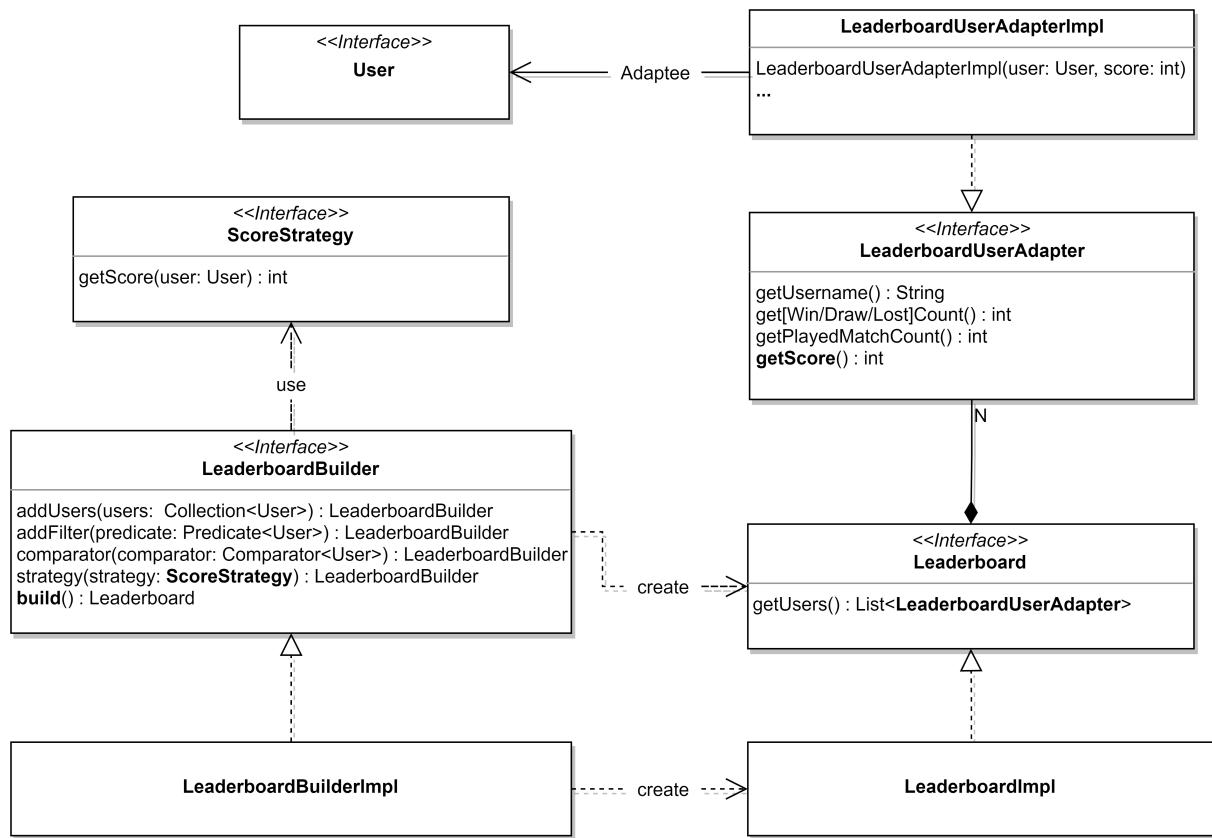


Figura 2.9: Leaderboard: creazione e contenuto

## 2.2.2 Mazzoli Alessandro

### Player

Durante la fasi di analisi, si è deciso di dividere l'entità `User` dall'entità `Player`, che ricoprono due ruoli differenti. La prima, trattata in un altro paragrafo apposito, gestisce l'utente che accede all'applicazione, mentre la seconda si occuperà del giocatore effettivo durante la partita. Il `Player` avrà una referenza allo `User` a cui è legato.

Nel gioco degli scacchi ogni player è identificato da un colore, che nel nostro caso è rappresentato dalla enum `PlayerColor`. Questo colore nella versione base degli scacchi può essere o bianco o nero, tuttavia avendola gestita come enumeration questo permette l'estensibilità ad un numero più elevato di colori in modo da gestire partite con un numero più elevato di giocatori (ad esempio potrebbe esistere una variante degli scacchi che si gioca in 4).

Per la generazione dei pezzi è stato scelto il pattern **Factory**, infatti ogni player ha dentro di sé una `PieceFactory`, che verrà usata per generare i propri pezzi, in modo da avere in essi una referenza al colore.

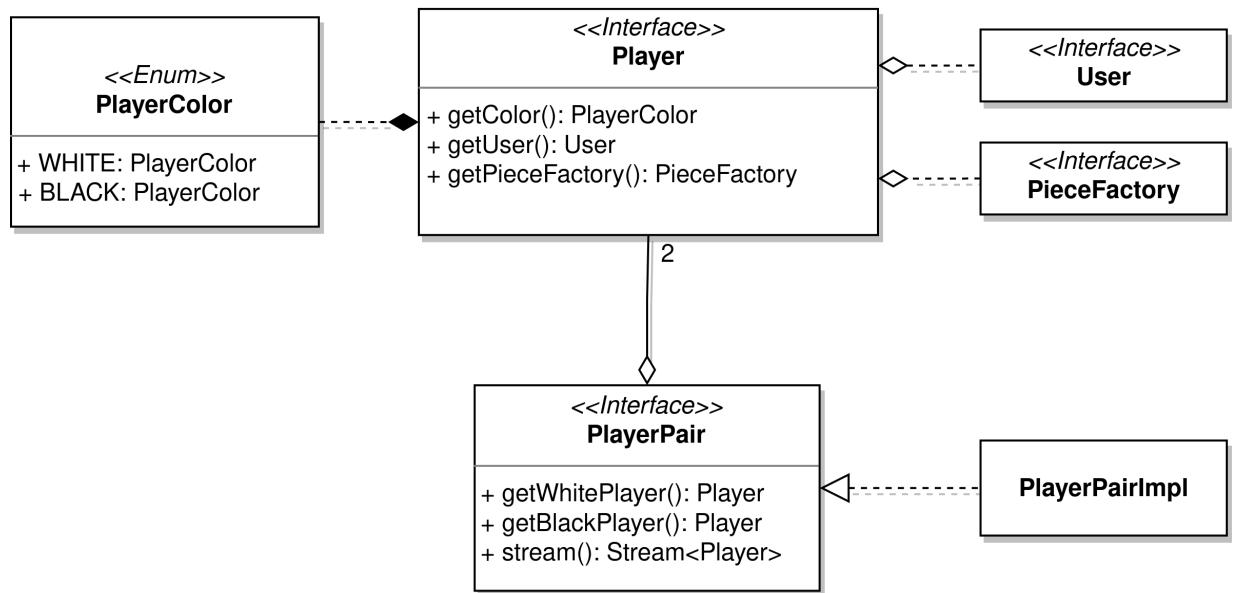


Figura 2.10: Gestione dei Players

Dato che si è scelto di gestire solamente varianti che hanno due giocatori, si ha avuto la necessità di salvarsi sempre la coppia `whitePlayer` e `blackPlayer` in molte parti del motore scacchistico. Ho quindi deciso di creare una classe specifica per contenerli: la `PlayerPair`. Questa fornisce l'accesso al `whitePlayer` ed al `blackPlayer` tramite getter specifici. Inizialmente infatti si era optato di utilizzare una `Pair` generica, tuttavia si doveva usare

una convenzione di accesso, ovvero: facendo `getX()` veniva restituito il `whitePlayer` e con `getY()` veniva restituito il `blackPlayer`.

Questo tuttavia creava molta confusione ed il codice risultava opaco, mentre con l'utilizzo della classe `PlayerPair` è stata rimossa questa problematica.

## Game

L'applicazione, come già premesso, deve supportare più varianti scacchistiche, non solo la modalità classica.

Per permetterne un' efficace funzionamento abbiamo dovuto analizzare dal punto di vista teorico ciò che caratterizza e differenzia le varie componenti di una modalità di gioco scacchistica, giungendo ad un breakdown in componenti essenziali della stessa.

Questa modalità di gioco è stata identificata quindi dall'interfaccia `Game`, tenendo conto che esso è composto da 3 macro-componenti:

- un `GameType`, ovvero il tipo di gioco a cui stiamo giocando, definito tramite una enumeration, che avrà dentro di sè informazioni quali il nome e una breve descrizione.
- un `MovementManager`, che si occupa dell'effettuare i movimenti dei pezzi controllando la loro fattibilità.
- un `GameController`, il cui scopo è quello di conoscere lo stato della partita, rilevare scacchi, stalli e controllare se la partita è giunta ad una conclusione o meno.

Possiamo vedere quindi il Game come un wrapper racchiudente le componenti sopra citate.

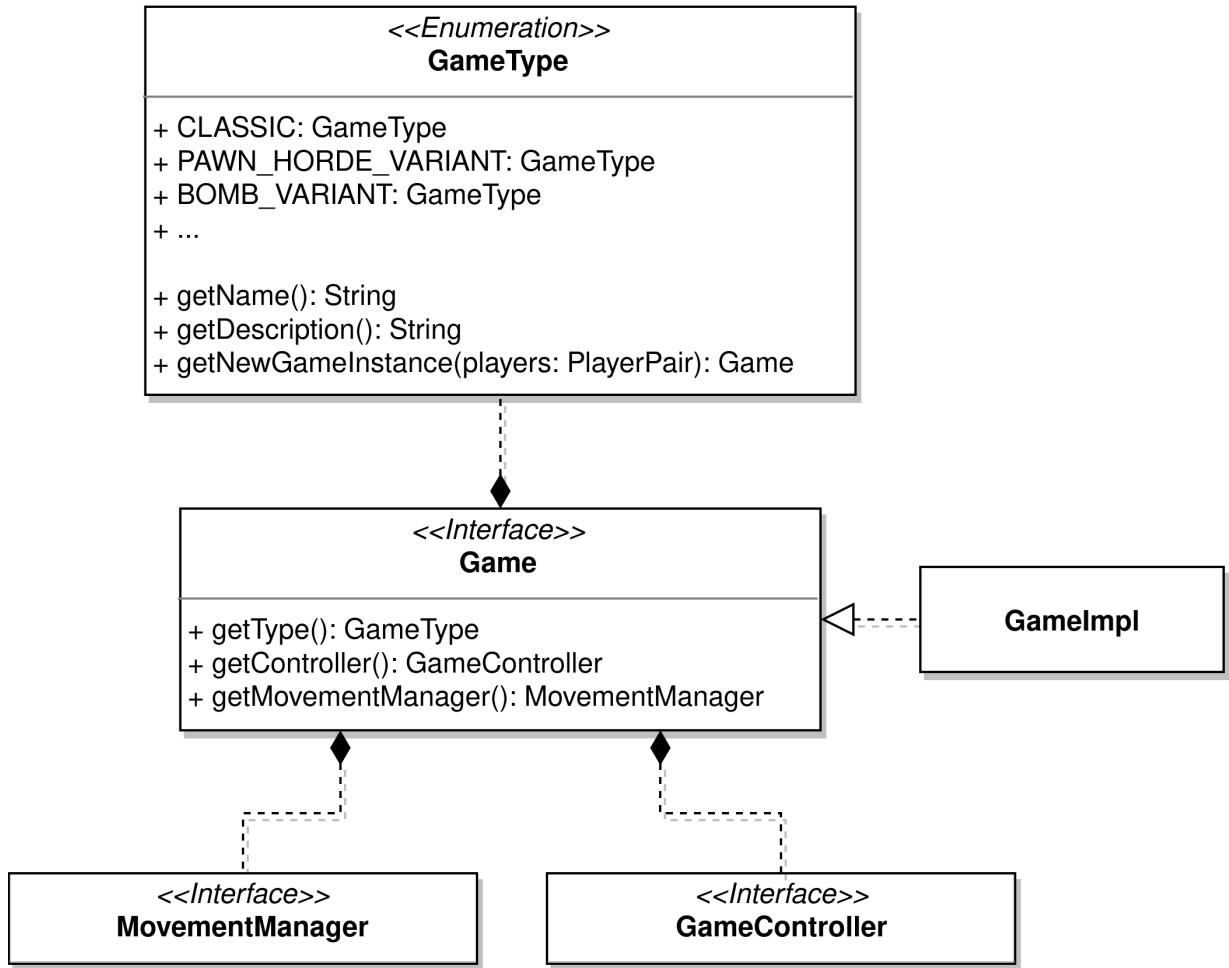


Figura 2.11: Gestione della struttura di una modalità di gioco degli scacchi con un alto livello di estensibilità

Per la creazione di un'istanza di `Game`, si è deciso di utilizzare il pattern **Factory** tramite l'interfaccia `GameFactory`. A questa è stato delegato il compito della creazione di istanze delle varie tipologie di `Game`.

Il suo compito è di generare le componenti adatte alla modalità di gioco richiesta, andando quindi a creare per ogni tipo di `Game` un'istanza di ogni sua componente interna citata precedentemente.

Si è scelto inoltre di utilizzare anche il pattern **Builder** nel `GameBuilder`. Nonostante i parametri del `Game` non siano molti, ho scelto di farne comunque uso perché ritengo se ne guadagni dal punto di vista della leggibilità del codice quando si tratta di creare un'istanza di un determinato oggetto.

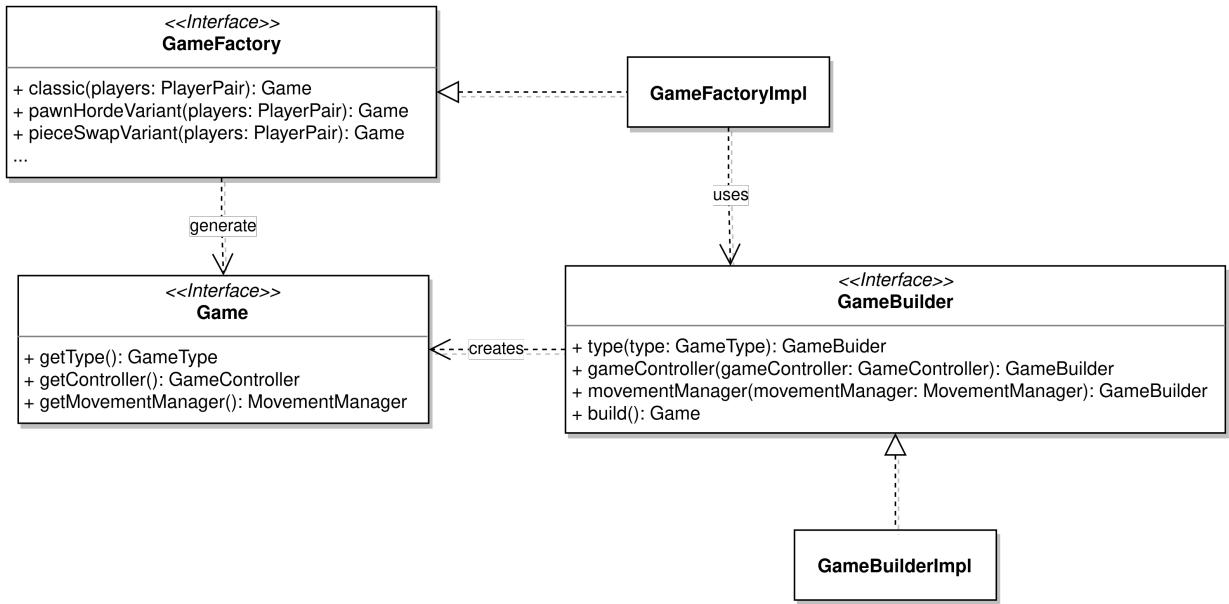


Figura 2.12: Creazione di un Game tramite pattern Factory e Builder

Si può notare la corrispondenza 1:1 tra una entry di `GameType` e di `Game`, infatti quello che ne risulta è che per ogni entry della Enum vi è un metodo nella `GameFactory` per creare un oggetto di quel tipo.

Sarebbe stato possibile encapsulare la creazione di questo oggetto totalmente all' interno dell'Enum `GameType`, tuttavia avrebbe portato ad un elevato numero di responsabilità delegate a quest'ultima e ne avrebbe aumentato notevolmente le dimensioni.

Quello a cui si è giunti è quindi un compromesso tra le due cose, poiché è molto comodo ottenere un' istanza di un Game dato il `GameType`, abbiamo aggiunto un metodo `getNewGameInstance(...)` all'interno dell'Enum che permette di ritornare una nuova istanza di un `Game` relativo a quel `GameType`. Questo metodo utilizza internamente una `GameFactory`, ed ogni entry di `GameType` deve specificare una strategia di creazione di quel determinato Game avendo come input la `GameFactory` e i players.

Questo è stato ottenuto utilizzando il pattern **Strategy** attraverso l'interfaccia funzionale `GameGenerationStrategy` il cui unico metodo permette di creare un' istanza di Game prendendo come parametri la `GameFactory` interna all'enum ed i players.

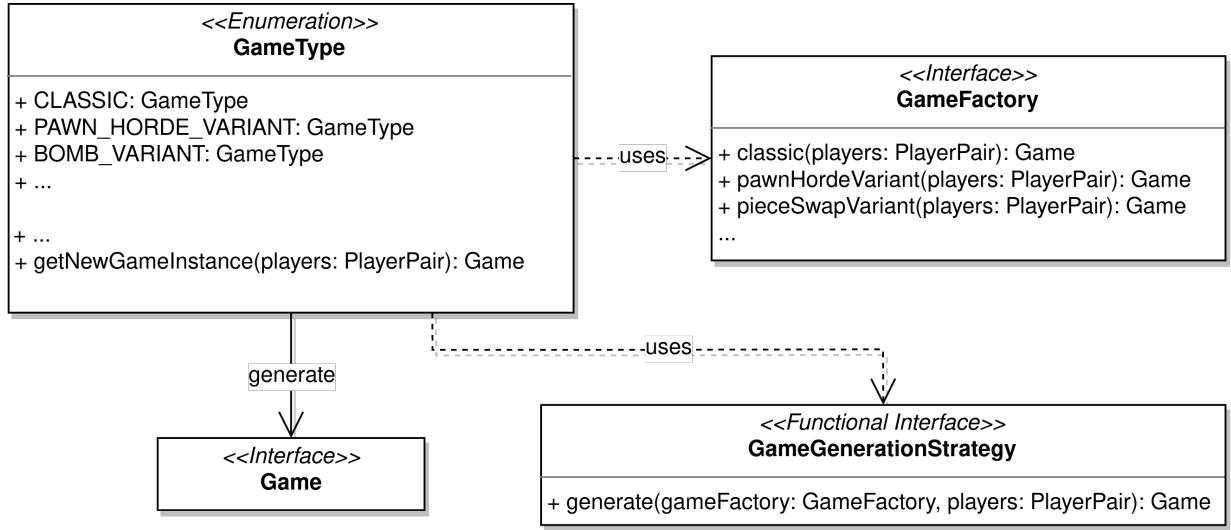


Figura 2.13: Generazione del Game in maniera agevole tramite pattern Strategy

## Match

Come già fatto con la gestione di `Player` e `User`, anche in questo caso ho deciso di differenziare l'entità `Game` dall'entità `Match`.

Il `Game`, come già detto precedentemente, è l'istanza del gioco attuale, comprensivo del suo progressivo stato durante la partita e delle sue regole, mentre il `Match` si può considerare un "wrapper" che gli aggiunge funzionalità.

Una partita vera e propria, infatti, non è rappresentata della sola entità `Game` ma necessita della presenza di altri componenti, tra i quali un `Timer` e la `History` della partita (la sequenza di mosse/stati eseguite).

Affidare tutte queste responsabilità all'entità `Game` avrebbe violato fortemente SRP e quindi si è optato per l'utilizzo dell'entità `Match`, che internamente ha l'istanza del `Game` ed inoltre tiene conto di tutte le altre componenti che compongono una vera partita di scacchi. Il controller della partita comunicherà quindi con questa entità, in essa troviamo infatti i metodi principali di gestione della partita, quali il metodo `move(...)` che permette di effettuare una mossa, il metodo `getPiecePossibleMoves(...)` che serve a restituire tutte le posizioni in cui può andare un determinato pezzo.

Anche in questo caso il `MatchStatus` è stato considerato diverso dal `GameStatus`, in modo che gestisca semplicemente se la partita è attiva oppure terminata. In questo modo solamente una volta che l'utente sa che la partita è finita potrà andare a controllare come questa si è conclusa tramite il metodo `getEndType(...)`, che utilizza l'Enum che caratterizza tutti i modi in cui può finire una partita.

Come si può notare, non abbiamo le stesse modalità di fine gioco del `Game`, infatti in questo caso oltre a `CHECKMATE` e `DRAW` vediamo comparire `RESIGN` e `TIMEOUT`, che come già detto sono feature esclusive del `Match` e sconosciute al `Game`.

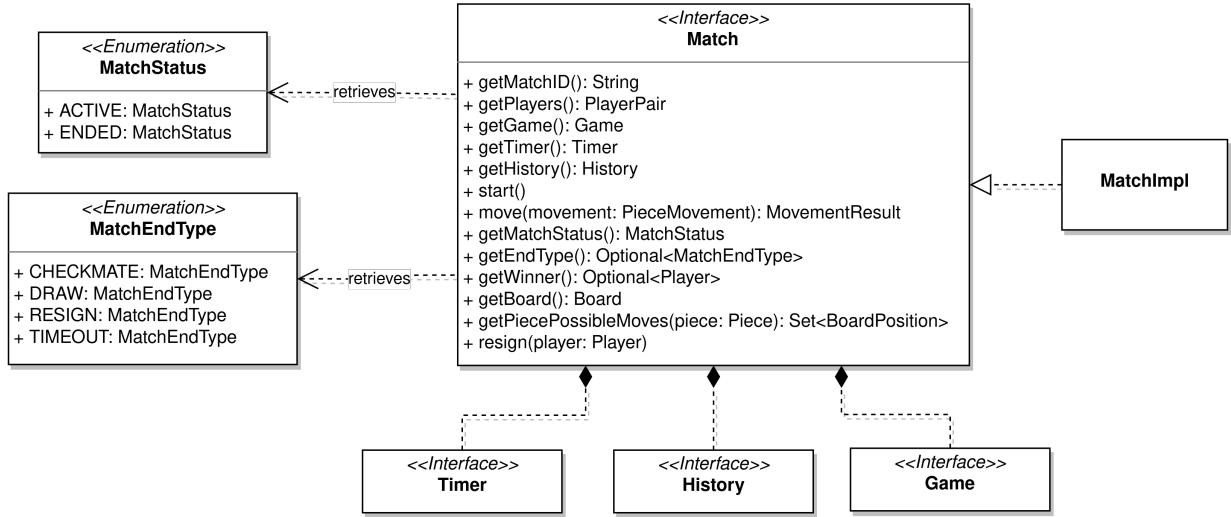


Figura 2.14: Struttura di un Match comprensivo delle sue componenti e stati

## Online Multiplayer

Feature a cui ho molto tenuto sin dall'inizio dello sviluppo è stata la possibilità di giocare da due client diversi tramite internet, poiché questo permette di godersi molto di più il gioco non dando la necessità di essere in due persone davanti allo stesso computer per poter fare una partita.

L'idea fin da subito era di utilizzare un mio server e creargli sopra un applicativo per la gestione della partita, tuttavia questo richiedeva di creare un database apposta per salvarsi le partite, gli utenti e soprattutto di trovare il modo di gestire in maniera efficiente la sincronizzazione tra i due client.

Nonostante fosse tutto più che fattibile questo lavoro avrebbe portato via un quantitativo enorme di ore, ma il vero problema sarebbe stato che nel caso di utilizzo di questa modalità si sarebbe andato a perdere l'utilizzo del motore scacchistico da noi scritto, poiché il gestore della partita sarebbe stato il server (era quindi inoltre necessario reimplementare un altro motore).

Non volendo rinunciare a questa feature ho deciso di creare una versione molto basilare di comunicazione via rete sfruttando il protocollo MQTT appoggiandomi al broker Mosquitto. Quello che fa quindi la modalità online è un handling della mossa utente e prima di inviarla al proprio motore la pubblica serializzata anche su un topic specifico tramite MQTT che comprende un base url 'jhaturanga/games/' concatenato al matchId. In questo modo l'altro client che è sottoscritto a quel topic riceverà la mossa, la deserializzerà e la invierà al proprio motore interno facendola sembrare una mossa fatta dall'utente, triggerando successivamente una callBack che avviserà che è stata fatta una mossa.

Quello che abbiamo è quindi che il `NetworkMatchManager` è colui che si occupa della creazione/join del match e dell'invio delle mosse all'altro client. Tuttavia per non dargli troppe responsabilità la gestione dell'invio dei dati e della connessione/disconnessione dal broker è stata affidata all'entità `MqttNetworkInstance` il quale contratto è quello di permettere i metodi basilari di comunicazione tramite MQTT quali:

- Connessione/Disconnessione
- Sottoscrizione ad un topic
- Invio di un determinato comando

La classe poi usata come implementazione di questo contratto è stata appunto la `MosquittoMqttNetworkInstance` che si serve del broker Mosquitto per la comunicazione.

Per l'invio e la coordinazione dei messaggi è stato inoltre necessario trovare un modo per capire che cosa si stesse inviando/ricevendo e quindi si è sviluppata un entità che funziona da DTO(Data Transfer Object) mediante l'utilizzo della classe `NetworkMessage`, che mi permette di incapsulare i dati da inviare memorizzando anche l'id di chi lo ha inviato. Quest'ultimo è servito poiché entrambi i client sono sia sottoscritti al topic e contemporaneamente ci pubblicano e quindi chi invia un messaggio lo riceve anche. Avendo quindi a disposizione l'id di colui che invia posso filtrare i pacchetti in arrivo in modo da avere solo messaggi dell'altro client.

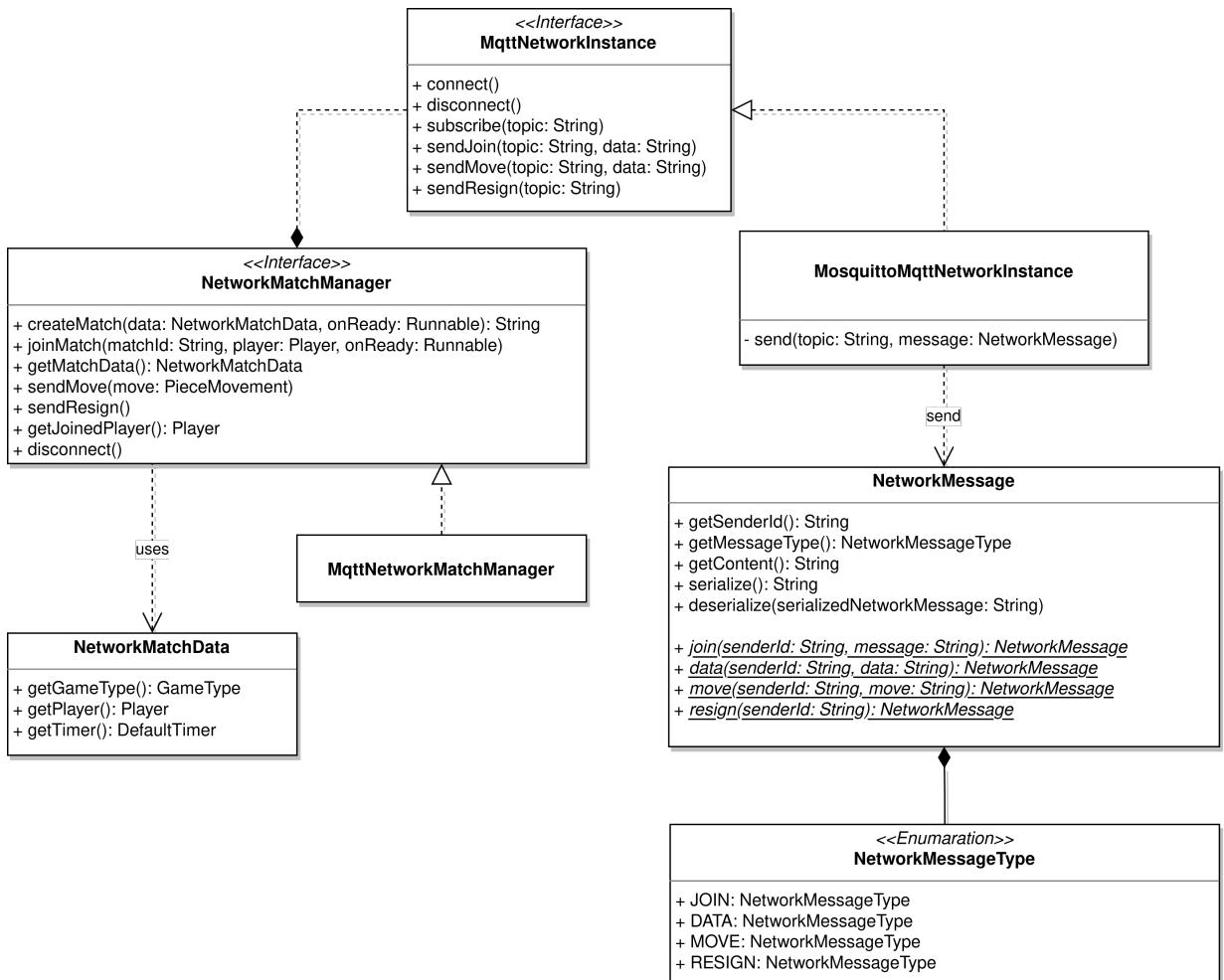


Figura 2.15: Struttura per la gestione della comunicazione via internet tramite protocollo MQTT

La gestione quindi della modalità online ad un più alto livello è ottenuta tramite l'utilizzo del pattern **Decorator** mediante l'entità **OnlineMatch**. Essa infatti implementa il contratto **Match** e al suo interno ha un'istanza di **MatchImpl** a cui delega i principali metodi che non richiedono l'invio di messaggi online, mentre quando viene fatta una mossa prima di inviarla al proprio **Match** interno utilizza il **NetworkMatchManager** per inviare questa mossa anche all'altro client.

Successivamente invia la mossa anche al proprio **Match** e restituisce il risultato della mossa all'utente.

Essendo necessario avvisare l'utente del ricevimento di una mossa è stata utilizzata una "callback" mediante la functional interface **MovementHandler** che permette al controller di specificare cosa fare quando viene ricevuta una mossa. Infatti quando una mossa viene

ricevuta viene inviata al Match interno e ne viene salvato il risultato, questi dati vengono poi passati alla chiamata della callback in modo da poter informare che mossa è stata fatta e che risultato ha avuto.

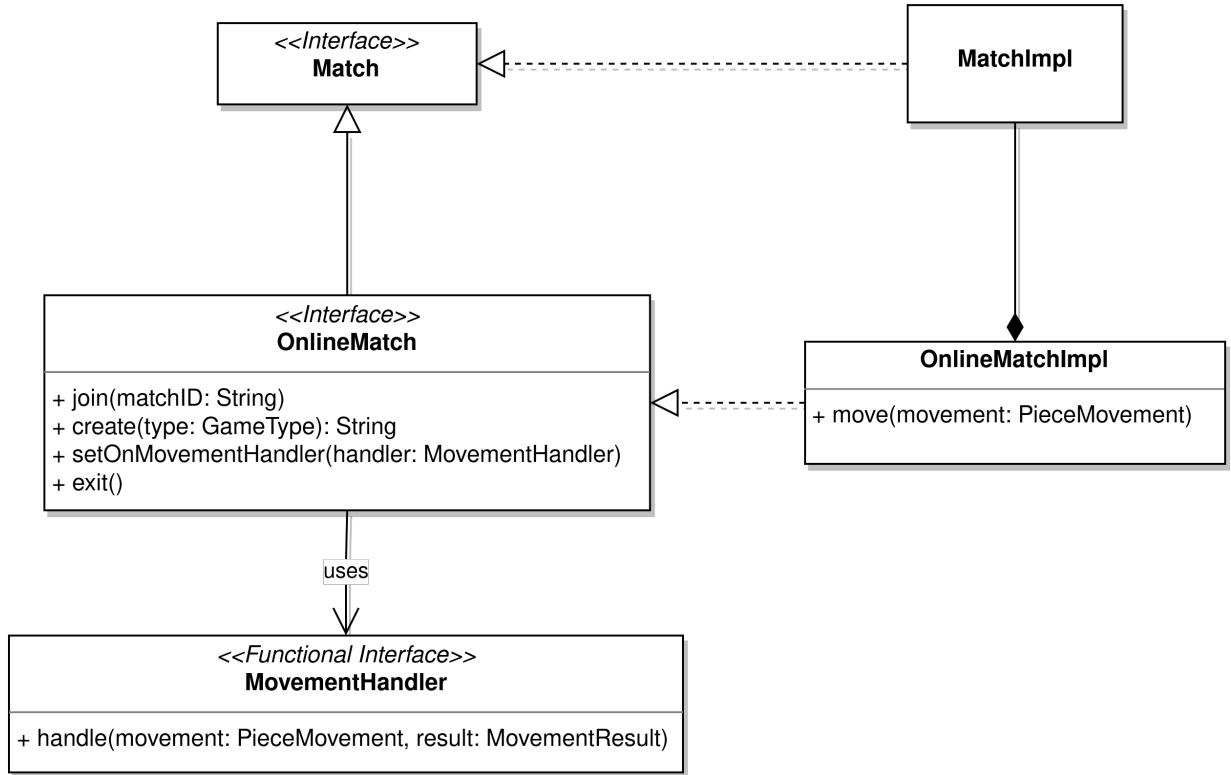


Figura 2.16: Gestione del Match online tramite pattern Decorator

## Graphical Board

Una volta terminata la mia parte, nel tempo libero, mi sono dedicato al refactoring di quella che era la board a livello grafico che era un po il "core" della parte visiva. Essa doveva gestire una quantità molto elevata di funzionalità, doveva permettere la visualizzazione dei pezzi, possibilità di movimento tramite trascinamento, navigazione della storia delle mosse tramite digitazione da tastiera e tanto altro...

Quello che ne risultava è che la classe della board era molto grande e poco facilmente gestibile, inoltre avendo vari tipi di board diverse queste erano ottenute tramite un copia/incolla di un'altra board opportunamente modificata poiché non erano riutilizzabili.

Tuttavia ciò violava fortemente DRY.

Ho quindi pensato di fare un breakdown di queste ultime per capire che cosa avevano in comune costruendo quindi quella che è la gerarchia della board. Ho di conseguenza estratto

quello che erano le funzionalità comuni a tutte le board mediante la classe **GraphicalBoard**, da quest'ultima prendono vita tramite estensione la **ReplayBoard** e la **MatchBoard**.

La **ReplayBoard** è una board che viene utilizzata per la navigazione di una partita salvata, non è quindi necessario avere nessun tipo di movimento dei pezzi da parte dell'utente, mentre è necessario averlo nella **MatchBoard**, la quale viene utilizzata per giocare una partita. Inoltre essa ha molte altre funzionalità quali la possibilità di evidenziare delle celle e le possibili mosse dei pezzi.

Un ulteriore estensione è la **OnlineMatchBoard** che deve mettere un vincolo ai movimenti possibili, ovvero l'utente locale che gioca online non può muovere i pezzi del giocatore avversario.

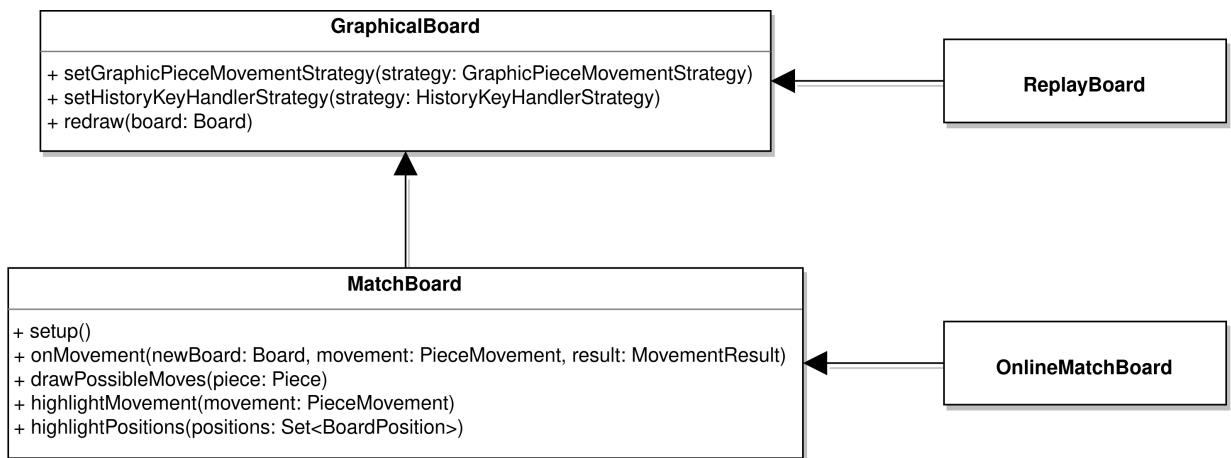


Figura 2.17: Gerarchie delle varie Board a livello grafico

Molto importante è stata la gestione dei movimenti dei pezzi a livello di interfaccia. Per implementare questa feature ho deciso di utilizzare il pattern **Strategy** che mi ha permesso di definire un contratto che le varie strategie di movimento dei pezzi dovevano implementare.

Ciò è stato ottenuto tramite l'interfaccia **GraphicPieceMovementStrategy** che rappresenta appunto la strategia di movimento grafico dei pezzi, il quale si compone di 3 eventi:

- Un pezzo viene cliccato
- Un pezzo viene trascinato
- Un pezzo viene rilasciato

Ciò mi ha permesso di definire appunto una struttura generale del contratto di movimento dei pezzi a livello grafico e ne sono giunte di conseguenza le implementazioni per una normale partita tramite la **NormalMatchGraphicPieceMovementStrategy** e la sua derivata **OnlineMatchGraphicPieceMovementStrategy** che come già detto precedentemente non

deve dare all’utente locale la possibilità di muovere i pezzi dell’avversario. Poichè questa strategia di movimento dei pezzi è stata resa parte dalla classe base **GraphicalBoard**, ovvero tutte le board la hanno, è stata necessaria e molto semplice la creazione della classe **NonMovableGraphicPieceMovementStrategy** che viene usato nelle board come la **ReplayBoard** in cui il movimento dei pezzi non è permesso.

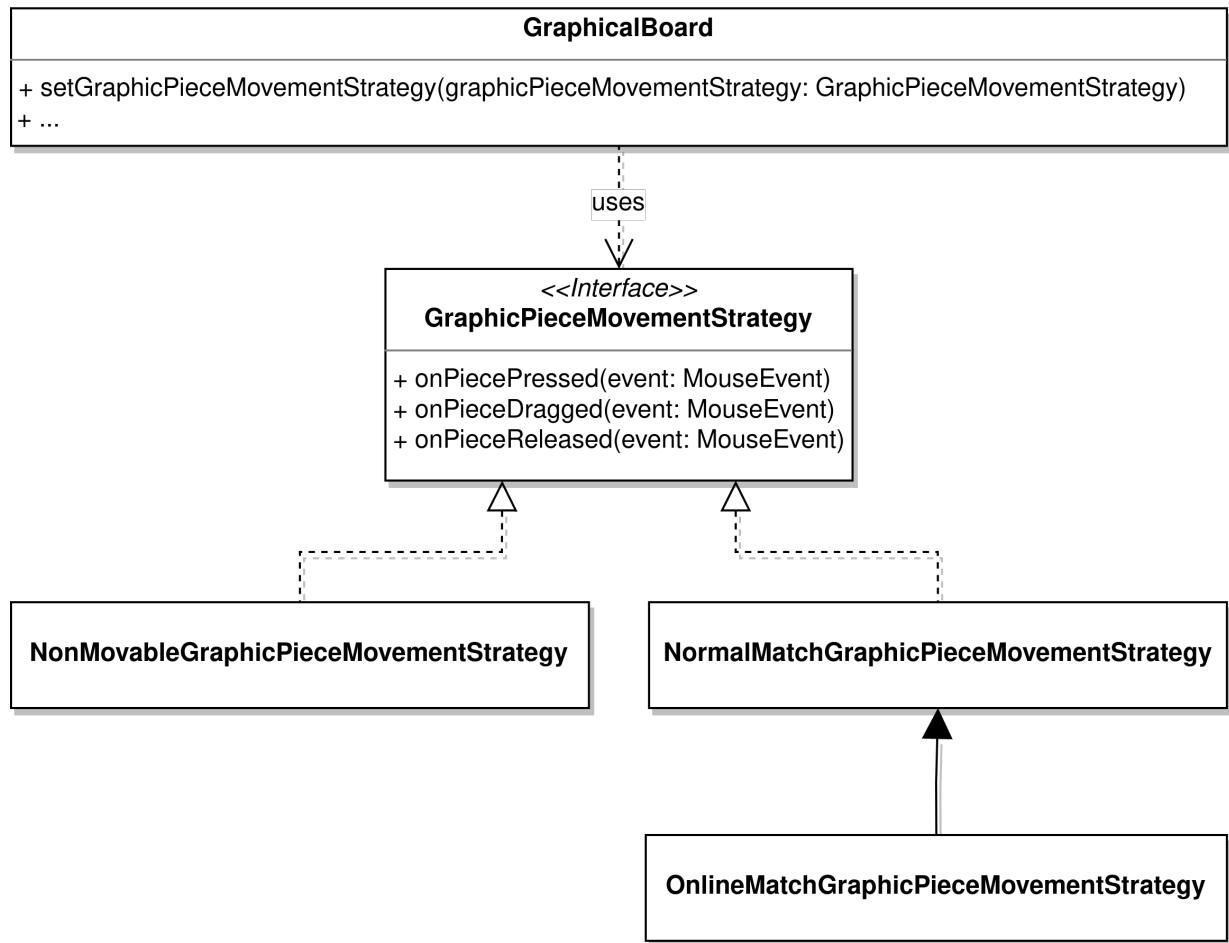


Figura 2.18: Strategia per il movimento dei pezzi a livello grafico

Altra necessità che molte board avevano in comune era la possibilità di navigare le mosse precedenti. Anche in questo caso ho deciso di utilizzare il pattern **Strategy** tramite la **HistoryKeyHandlerStrategy**, ovvero la strategia che definisce cosa fare quando viene premuto un tasto sulla tastiera da parte dell’utente.

È stato dunque molto semplice gestire analogamente alla **GraphicPieceMovementStrategy** due semplici strategie di navigazione della **History**, una che prevede una normale navigazione tramite le frecce (**NormalHistoryKeyHandlerStrategy**) e un’altra che invece

non rende possibile la navigazione (`NonNavigableHistoryKeyHandlerStrategy`) adottata in quelle board che non hanno bisogno della navigazione della storia delle mosse.

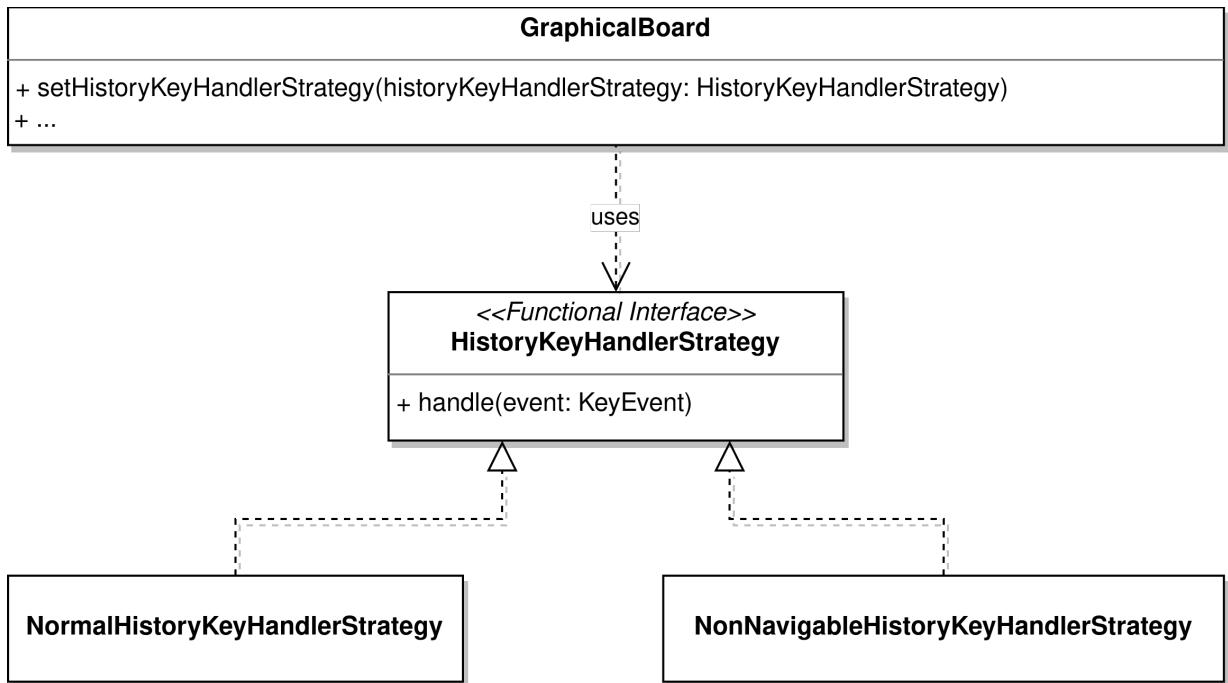


Figura 2.19: Navigazione della History di una partita mediante pattern Strategy per l'handling del click dei tasti

## 2.2.3 Tommaso Patriti

### Replay

Replay serve per mantenere uno storico con la cronologia delle mosse delle partite giocate con Jhaturanga, è implementato in tutte le sue versioni: classica, varianti, online e scacchiera personalizzata.

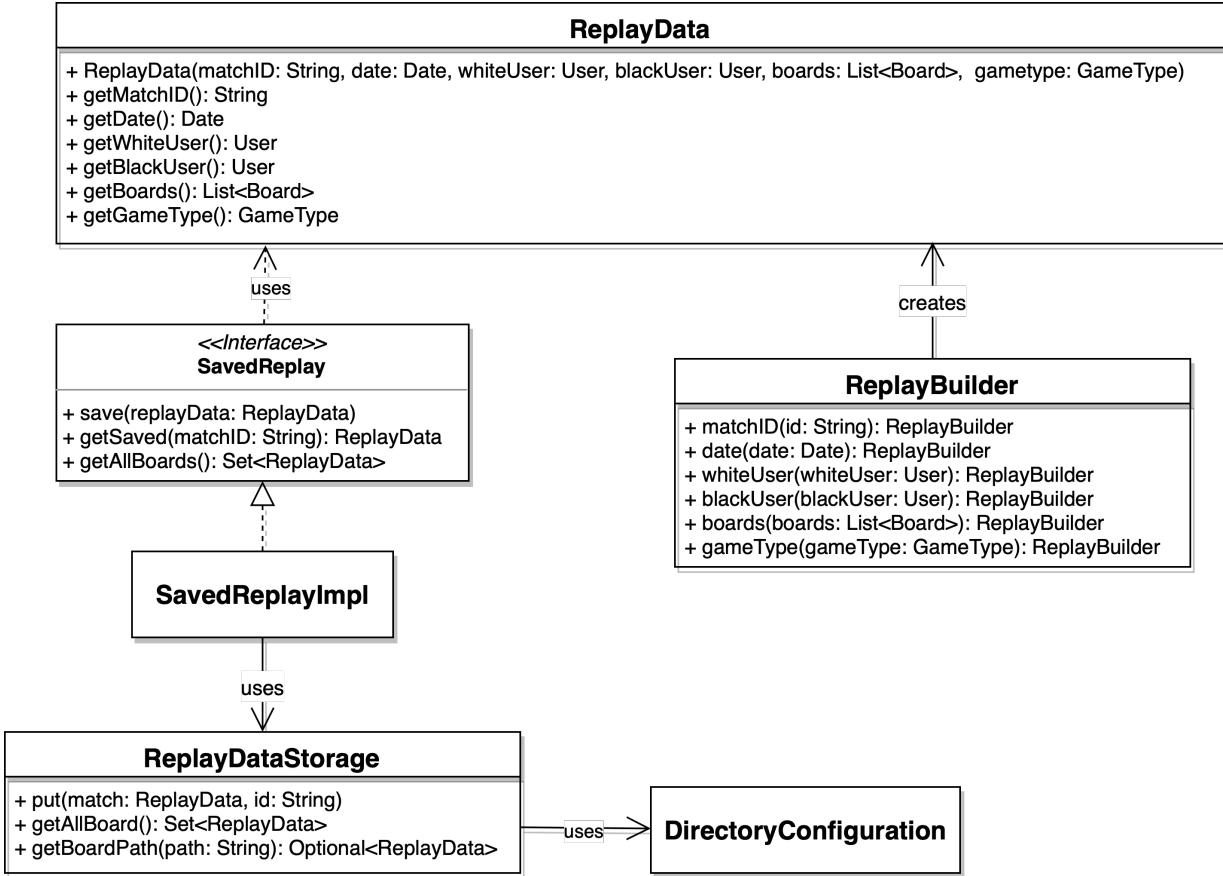


Figura 2.20: Gestione salvataggio partite e Replay

**ReplayData** è una classe che contiene tutte le informazioni necessarie per il salvataggio di una partita.

Considerando i numerosi campi necessari alla sua creazione, ho deciso di implementare il pattern **Builder**.

**ReplayData** è un oggetto serializable in maniera tale da poter essere salvato e riutilizzato agevolmente.

**ReplayDataStorage**, si occupa di comunicare con la cartella .jhaturanga utilizzata per il salvataggio dei file permanenti dell'applicazione.

La gestione della cartella .jhaturanga è implementata in DirectoryConfiguration, che è condivisa da tutti gli oggetti che hanno necessità di accedere a file salvati nel computer. .jhaturanga è una cartella che si trova nella home directory in cui viene eseguita l'applicazione. SavedReplay si occupa di gestire la comunicazione tra ReplayDataStorage e il resto dell'applicazione, in maniera tale da avere sempre un corretto utilizzo dei file.

## Timer

Il **Timer** è un oggetto che serve per la gestione del tempo durante una partita di scacchi.

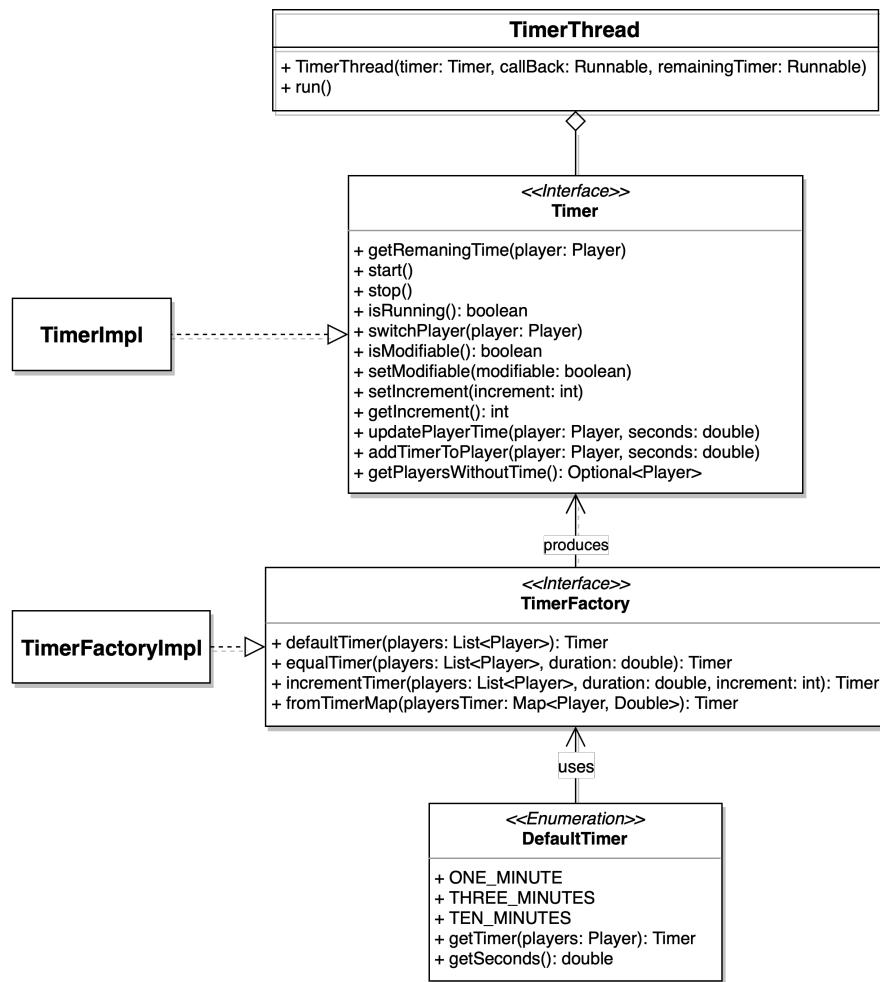


Figura 2.21: Gestione Timer

Il **Timer** è un oggetto che tiene in memoria la lista dei giocatori e il tempo loro rimanente. È stato implementato il pattern **Factory** per poter avere a disposizione varie tipologie di distribuzione del tempo. Visto che negli scacchi si usano dei timer ricorrenti, ho deciso

di dichiarare questi ultimi all'interno dell'enum DefaultTimer. Il Timer deve funzionare indipendentemente rispetto alle operazioni svolte sulla scacchiera, quindi serve un thread separato per gestirlo. Inizialmente quest'ultimo era direttamente implementato nella view, ma ciò portava side-effects nel suo utilizzo quindi ho implementato TimerThread che estende Thread, e richiama l'applicazione ogni volta che aggiorna il suo stato e quando uno dei partecipanti finisce il tempo. Questo viene fatto tramite la FunctionalInterface Runnable.

## Audio

Semplice classe con metodi statici per gestire l'audio

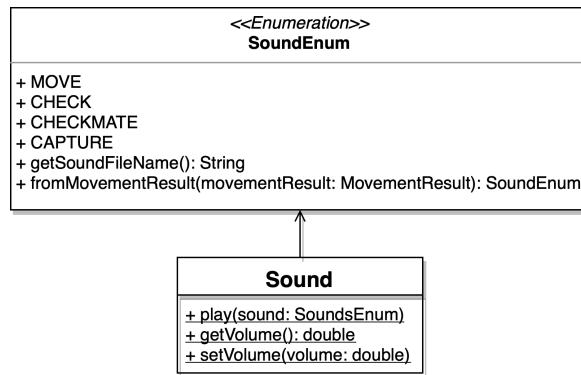


Figura 2.22: Gestione del suono

## Style

La gestione degli stili è stata sviluppata affinché l'applicazione ne abbia di default da poter selezionare. È implementata anche la possibilità di aggiungerne di nuovi, potendo quindi personalizzare quelli esistenti.

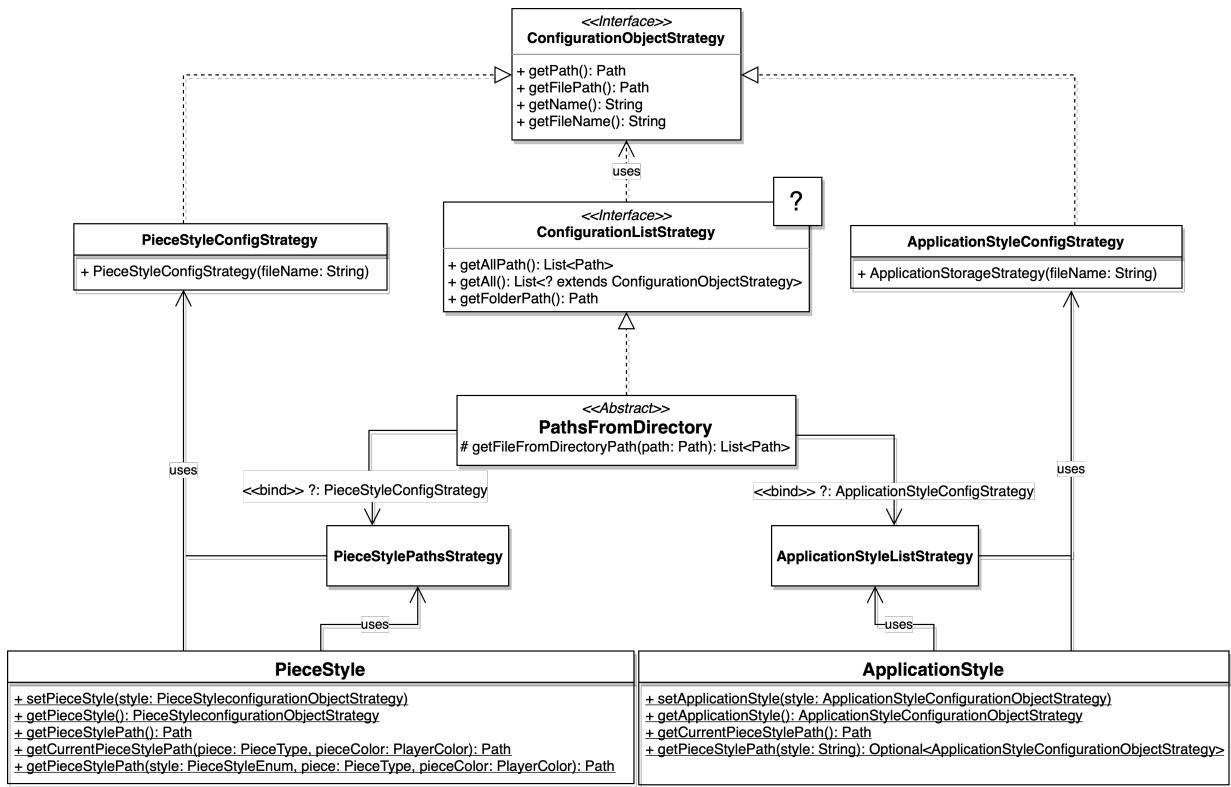


Figura 2.23: Gestione degli stili

**PieceStyle** gestisce lo stile di tutti i pezzi della scacchiera cambiando il percorso della cartella in cui si trovano. In caso non ci siano stili preferiti, la classe ne prevede uno di default.

**ApplicationStyle** gestisce i fogli di stile (CSS) usati da JavaFx. In caso non ci siano stili preferiti, la classe ne prevede uno di default. Queste due classi hanno una gestione simile, quindi a prima vista sarebbe possibile andarle a gestire con un eventuale **Template Method** implementato per mezzo dei tipi generici. Ho deciso di non seguire questa strada poiché l'utilizzo delle due classi è differente e le righe di codice in comune sono esigue, quindi, andandone a generalizzare l'utilizzo, avrei sacrificato la chiarezza del codice e portato ad un limitato incremento di flessibilità. Queste due classi per funzionare hanno bisogno di avere a disposizione una lista di stili. Inizialmente li ricavavano da una classe Enum, ma questo comprometteva l'estensibilità del codice, quindi ho deciso di eliminare queste ultime e creare delle liste di stili dipendenti dalle risorse disponibili in .jhaturanga. All'interno di quest'ultima in res/piece possiamo aggiungere cartelle contenenti le immagini degli scacchi, e in /res/css/themes, file css, per lo stile dell'applicazione.

Per gestire queste risorse, ho deciso di utilizzare **Strategy** in due punti:

- **ConfigurationObjectStrategy** serve per creare degli oggetti che rappresentano gli

stili, quindi, serve a contenere i loro percorsi, e dei metodi utili alla manipolazione specifica di questi ultimi.

- *ConfigurationListStrategy* serve a contenere tutte le istanze disponibili di *ConfigurationObjectStrategy*, quindi, in pratica, la lista degli stili disponibili. *PathForm-Directoy* è una classe astratta che si frappone tra *ConfigurationObjectStrategy* e le sue implementazioni, e fornisce metodi utili alla lettura e scrittura delle risorse.

## Setting

Setting serve per gestire le personalizzazioni scelte dall'utente e si occupa di mantenerle sul computer in maniera tale da averle permanentemente salvate.

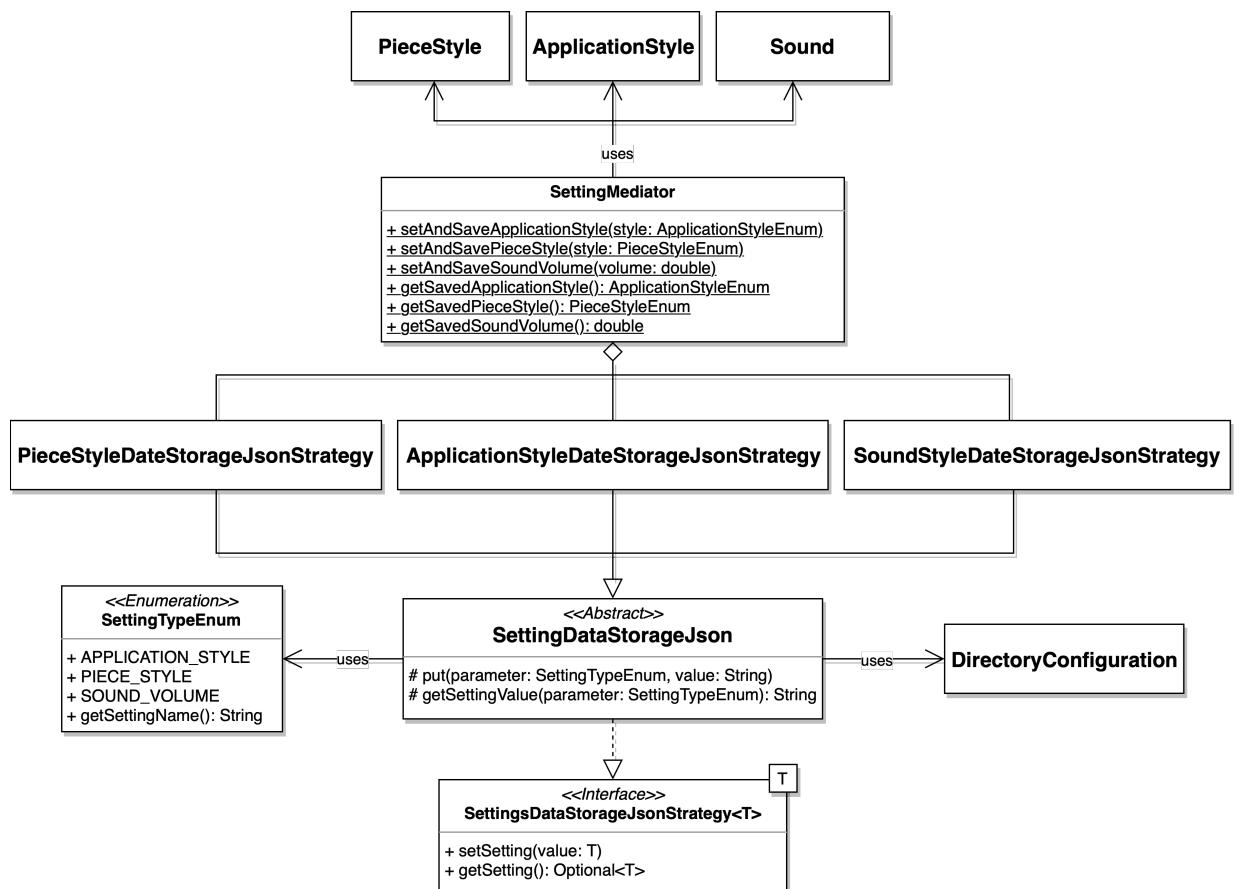


Figura 2.24: Gestione dei setting

L'obiettivo di questa parte di progetto è quello di poter salvare permanentemente le imo-

stazioni preferite dall’utente.

Ho deciso di mantenere separata la gestione del salvataggio dalla parte di configurazione. SettingDataStorageJsonStrategy è un’interfaccia che serve per l’implementazione del pattern **Strategy**. Ho stabilito di usare strategy in quanto le operazioni da fare sul file json sono sempre uguali, cambia solo l’interpretazione dei dati al momento della lettura e della scrittura. Pertanto, ad ogni configurazione corrisponde una classe che ne permette il salvataggio, ma che opera in maniera indipendente dalla classe di configurazione vera e propria.

Poiché in questo modo l’entità per il salvataggio delle configurazioni e le entità per la gestione delle configurazioni erano disgiunte, ho deciso di implementare settingMediator che usa il pattern **Mediator** per avere un’unica classe statica a cui accedere dal resto dell’applicazione che va a gestire l’interazione tra gestione e salvataggio delle impostazioni. Inoltre ho deciso di frapporre una classe astratta SettingDataStorageJson per avere dei metodi per la gestione del file Json in tutte le classi che hanno stretto un contratto con SettingDataStorageStrategy La lista dei parametri gestiti dai settings è fornita da SettingTypeEnum. Inizialmente SettingDataStorageStrategy per evitare problemi di scrittura nel json, era implementata per mezzo di stringhe, ma, per rendere il codice più espressivo e immediato da usare, ho deciso di optare per i generici, in modo tale che nelle ramificazioni di Strategy si possa decidere in libertà come gestire il parametro che verrà poi scritto e letto nel file json.

## 2.2.4 Scolari Stefano

Fin dalle fasi di analisi uno tra i nostri principali obiettivi era quello di creare un applicativo in grado di gestire un numero molto elevato di varianti scacchistiche.

La sfida principale da me affrontata è stata quindi la realizzazione di un'architettura di base il quanto più estendibile possibile, sia dal punto di vista dei movimenti dei singoli pezzi gestiti dalla **PieceMovementStrategies**, sia per quanto riguarda le regole che governano le varie modalità di gioco, queste ultime gestite sia dal **GameController** che dal **MovementManager**.

Adotterò un approccio bottom-up per descrivere il design dell' architettura da me implementata, tracciando e descrivendo quelli che sono i layer attraverso cui ho deciso di stratificare il motore scacchistico di base.

### PieceMovementStrategies

Per la realizzazione del concetto di strategia di movimento dei singoli pezzi ho pensato non fosse corretto rendere questa una caratteristica intrinseca dei **Piece**, infatti, proprio come nelle partite di scacchi reali, chi decide come muovere un pezzo è un'entità superiore ed esterna al **Piece** stesso.

Per la generazione dei **Piece** ho utilizzato il pattern **Factory**.

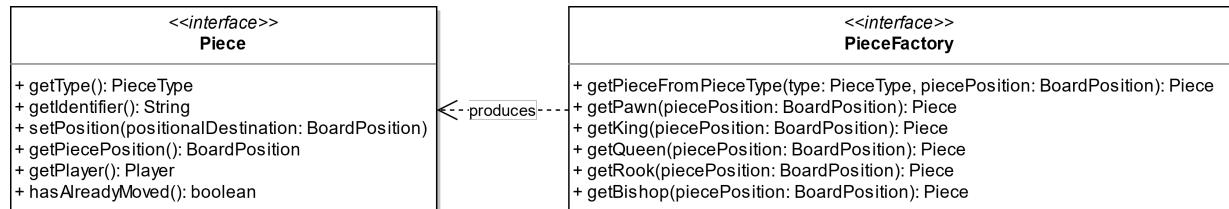


Figura 2.25: Struttura dell'entità **Piece** e sua rispettiva Factory

Ho invece deciso di utilizzare il pattern **Strategy** per la realizzazione delle **MovementStrategy** dei vari pezzi.

**MovementStrategy** risulta essere una Functional Interface, avente come unico metodo `getPossibleMoves(...)`, che prende come argomento la **Board** attuale, e ritorna l'insieme delle possibili destinazioni.

Da notare che la **MovementStrategy** è in grado di fornire una strategia di movimento generale, statica rispetto ai concetti ed alle regole di gioco. Gli unici controlli "dinamici" apportati sono che un pezzo non può scavalcare/catturare un pezzo alleato oppure superare un pezzo nemico lungo la traiettoria definita da uno **UnaryOperator** sulle **BoardPosition**. Concetti e regole più specifiche sono gestiti, come detto precedentemente, in parte dal **MovementManager** ed in parte dal **GameController**.

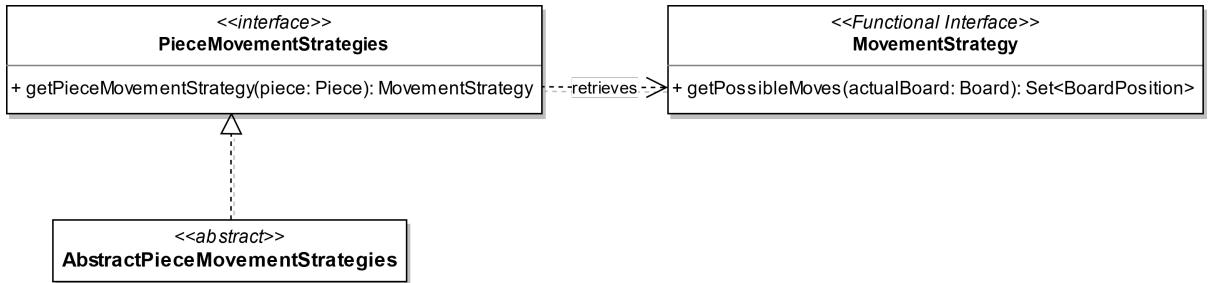


Figura 2.26: Struttura della `PieceMovementStrategies`

Il calcolo della strategia di movimento di un `Piece` viene eseguito a run-time, l'unico modo per interracciarci dall'esterno con la `PieceMovementStrategies` risulta essere il metodo `getPieceMovementStrategy(...)`, che prendendo un pezzo in input è in grado di restituirne la corretta strategia di movimento, sulla quale andrà eventualmente invocato il metodo dell' interfaccia funzionale per conoscere le effettive destinazioni possibili secondo quella specifica strategia.

Per permetterne il funzionamento ho utilizzato in `AbstractPieceMovementStrategies` un **Template Method**, che è proprio `getPieceMovementStrategy(...)`. Quest' ultimo, in base al `PieceType`, deve scegliere quale metodo specifico invocare per calcolarne la corretta Strategy di movimento. Queste singole strategie di movimento dei singoli pezzi non sono univoche, ma variano a seconda della modalità di gioco. Sono infatti definite inizialmente da metodi abstract in `AbstractPieceMovementStrategies` e per questo ho dovuto ricorrere al Template Method sopracitato.

Ho inoltre deciso di utilizzare questa classe astratta in modo tale da racchiudere in essa le caratteristiche comuni a tutte le varianti, come lo è il metodo `getDestinationsFromFunction(...)`, responsabile della generazione di tutte le mosse possibili a partire da alcuni parametri in ingresso.

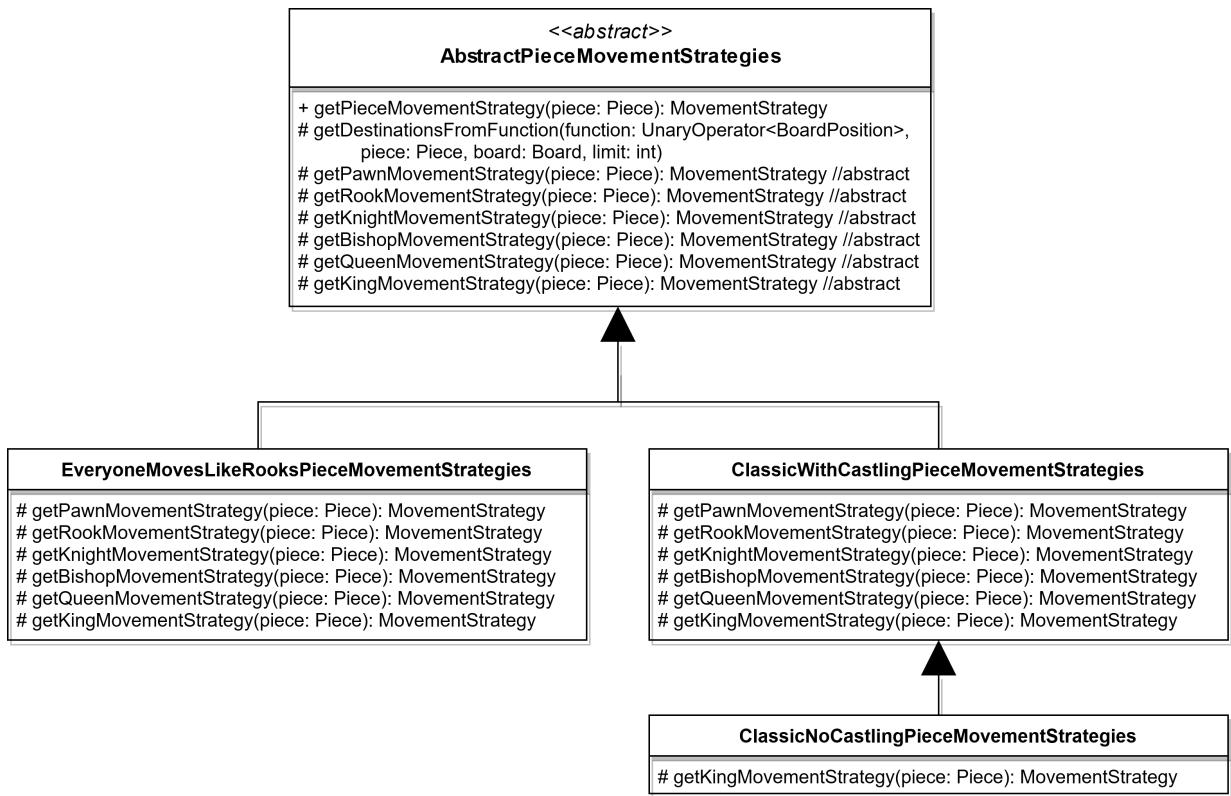


Figura 2.27: Interfacciamento con le strategie di movimento dei pezzi attraverso il Template Method `getPieceMovementStrategy()`

Aggiungere nuove varianti risulta semplice e veloce. La modalità classica degli scacchi rifà le sue strategie di movimento alla **ClassicWithCastlingPieceMovementStrategies**. Per alcune varianti risulta comodo estendere quest'ultima, andando poi a fare l'Override di solo quelle specifiche strategie di movimento dei pezzi che si vogliono modificare, lasciando invariate le altre.

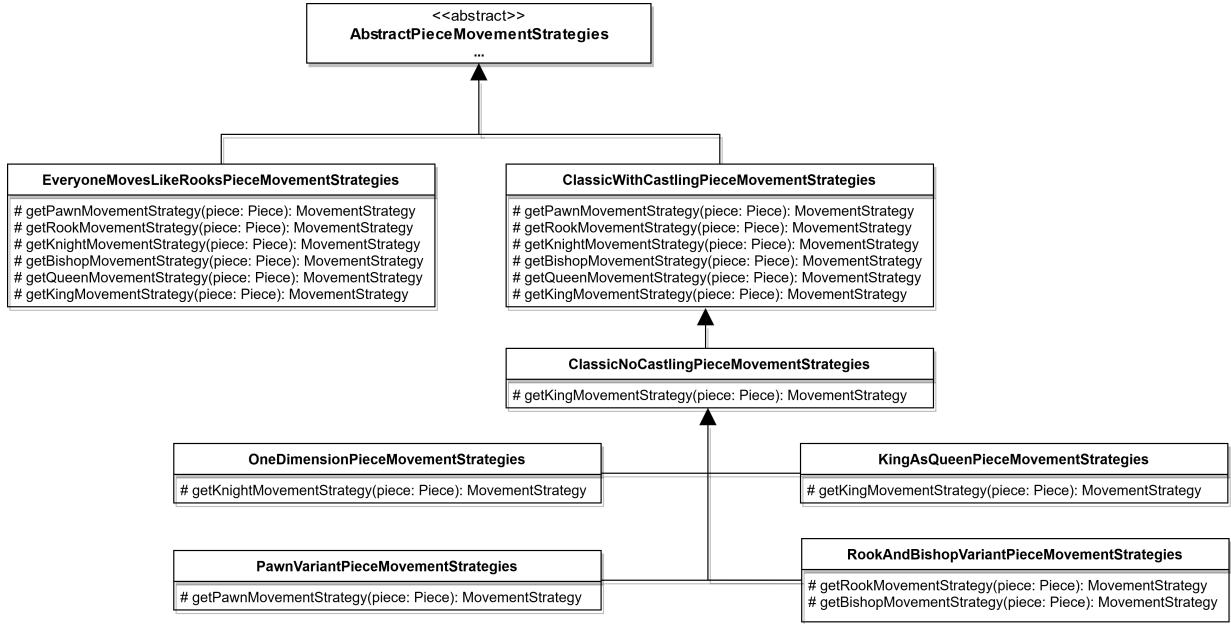


Figura 2.28: Implementazione delle diverse PieceMovementStrategies

Per altre invece, per le quali si intende modificare completamente tutte le strategie di movimento di tutti i pezzi, discostandole da quello che è la modalità classica, conviene invece estendere direttamente da `AbstractPieceMovementStrategies`.

`EveryoneMovesLikeRooksPieceMovementStrategies` ne è un esempio.

Essendo poi riuscito a catturare il modo attraverso cui sono individuate le possibili destinazioni per le varie `MovementStrategy` con `getDestinationsFromFunction(...)`, la generazione di queste risulta anche molto snella.

## Game Controller

Il `GameController`, come prima visto, è utilizzato soprattutto all'interno della `MovementManager`.

Ruolo del `GameController` è quello di implementare quelle che sono le regole della partita di scacchi, gestendo quindi i concetti di scacco, scacco matto, stallo e vittoria.

`MovementManager`, `PieceMovementStrategies` assieme al `GameController` possono essere considerate come la triade rappresentante il cuore del motore di gioco.

Il `GameController` è utilizzato quindi per conoscere lo stato attuale della partita dal punto di vista delle regole scacchistiche, appunto per questo è fondamentale all'interno del `MovementManager`.

Poichè il `GameController` è colui che analizza lo stato della partita è l'unico in grado di sapere se effettuare una mossa causerà uno scacco o meno. Questo controllo è eseguito con il metodo `wouldNotBeInCheck(...)`, che preso un `PieceMovement`, è in grado di valutare

se porterà o meno ad uno stato di scacco, portando a considerare quindi quest' ultima una mossa illegale(oppure no, in base alla variante).

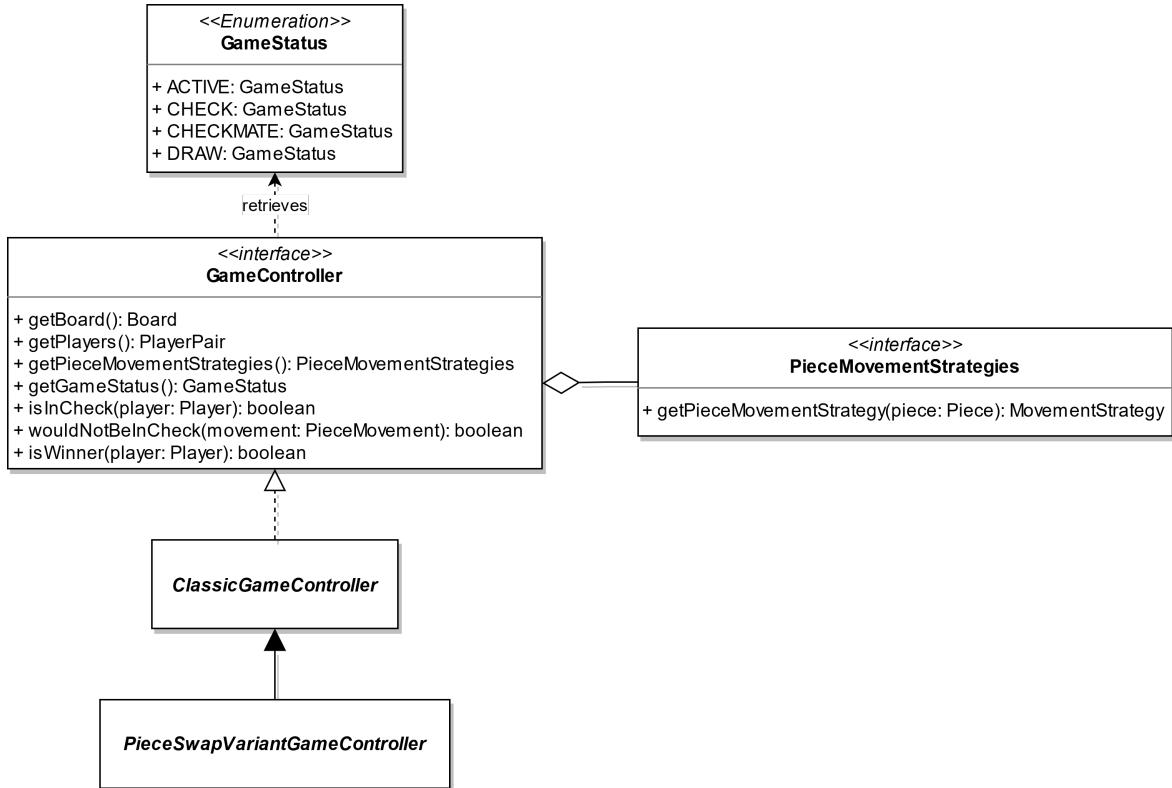


Figura 2.29: Struttura del GameController

Il **MovementManager** utilizza ampiamente questo metodo quando deve valutare se la mossa eseguita dall'utente sia o meno valida. Inoltre, il **GameController** fa ampio uso della **PieceMovementStrategies** per sapere come si possano muovere i pezzi sulla scacchiera, in modo tale da poter trovare eventuali mosse in grado di salvare un giocatore da una potenziale situazione di scacco matto.

Quasi tutte le modalità di gioco utilizzano la **ClassicGameController**, che esprime quelle che sono le regole della modalità classica degli scacchi. Unica eccezione è la variante **PieceSwapVariantGameController**, per la quale è stato necessario ridefinire la condizione di patta legata a "materiale insufficiente".

### **MovementManager**

Per poter facilmente conoscere la fattibilità o meno di una mossa eseguita, è stata necessaria la creazione di un **MovementManager**, il cui compito principale è quello di verificare la correttezza di una mossa e gestire i side-effect provocati dalla suddetta, comunicando poi

il risultato/conseguenza che questa ha provocato.

Per la realizzazione del **MovementManager** ho deciso di utilizzare il pattern **Facade**. Infatti, nonostante vi sia necessità di interagire con diversi oggetti, eseguendo altrettanti controlli per determinare la fattibilità della mossa passata, interagire con il **MovementManager** non risulta complesso, bensì "straightforward".

L'architettura definita può essere infatti vista come "layered": il **GameController** è utilizzato all'interno del **MovementManager**, e la **PieceMovementStrategies** all'interno del **GameController**. L'interfacciamento principale con "l'esterno" è il metodo **move(...)**. Inoltre, per evitare di dare troppe responsabilità alla sola **MovementManager** ho scelto di comporre quest'ultima di una **MovementHandlerStrategy**, che si occupa delle sole verifiche riguardo la correttezza/fattibilità della mossa passata. Ho favorito quindi la composition, catturando funzionalità tra loro affini tramite oggetti.

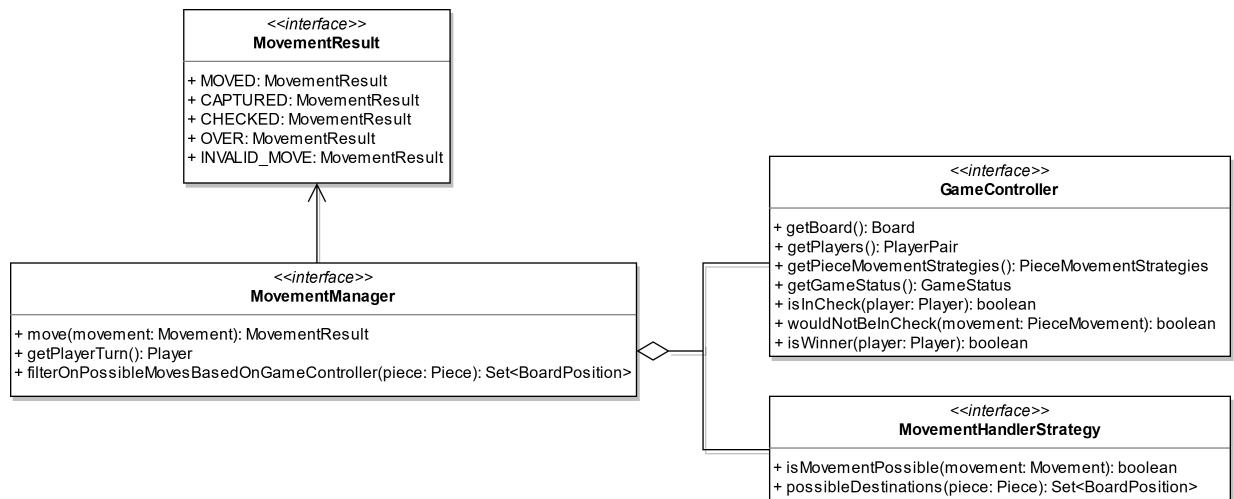


Figura 2.30: Gestione delle mosse attraverso il **MovementManager**

Una mossa, detta **Movement**, contiene le sole informazioni essenziali, quali posizione di origine e di destinazione. Dalla **BasicMovement** estende la **PieceMovement**, che in più contiene anche il **Piece** interessato dalla mossa ed un metodo per eseguire la mossa stessa, andando a modificare con una **SetPosition** la posizione attuale del **Piece** da **Origin** a **Destination**.

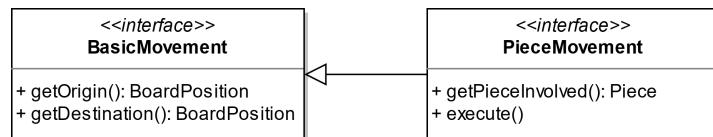


Figura 2.31: Focus di dettaglio sui **Movement**

Perchè il concetto di estendibilità non poteva fermarsi alle sole strategie di movimento, anche la maniera attraverso cui è gestita la fase di verifica e gestione delle conseguenze presenta una struttura facilmente estendibile e modificabile.

Con side-effect si intende, nel caso della modalità classica per esempio, la rimozione dei pezzi catturati, la promozione dei pedoni una volta raggiunta l'ottava traversa per i bianchi(oppure la prima per i neri) oppure lo spostamento della torre in caso di arrocco.

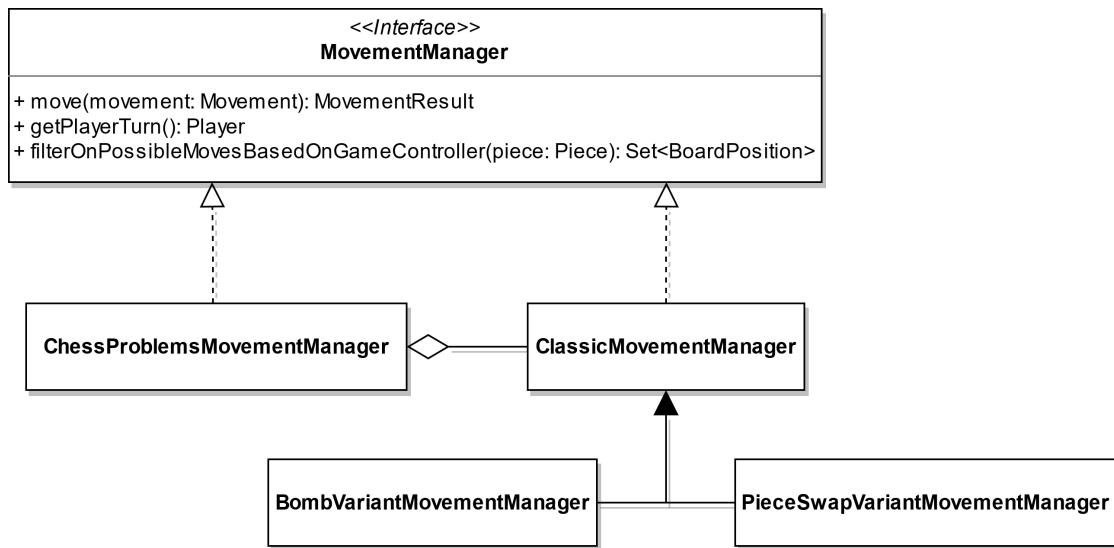


Figura 2.32: Implementazione dei vari MovementManager

Anche in questo caso è stato alquanto rapido e privo di difficoltà creare dei **MovementManager** alternativi a quello classico. Come la **PieceSwapMovementManager**, il cui side-effect particolare è quello di modificare secondo uno schema fisso il **PieceType** del pezzo appena mosso. Quando eseguita una mossa con una torre, questa si trasforma in un alfiere, quando mosso un alfiere, questo diventa un cavallo, e quest'ultimo quando mosso invece diventa una torre.

Oppure la **BombVariantMovementManager**, che come conseguenza ad una mossa ha la possibile "esplosione" del pezzo appena mosso, provocando la rimozione di tutti i pezzi che si trovino all'interno di un range randomico.

Anche l'implementazione dei problemi scacchistici necessitava di un proprio **MovementManager**, ma perchè la modalità di gioco che interessava i problemi era proprio quella classica, ed un **MovementManager** di quel tipo esisteva già, ho scelto di utilizzare il pattern **Decorator**. **ChessProblemsMovementManager** è un decorator del **ClassicMovementManager**. In più deve semplicemente verificare che la mossa effettuata sia quella corretta rispetto alla soluzione del problema scacchistico e come side-effect ulteriore deve eseguire la mossa dell'avversario una volta fatta quella corretta.

Questa architettura permette di definire modularmente molte varianti scacchistiche per ag-

gregazione delle diverse `PieceMovementStrategies`, `MovementManager` e `GameController` quando definito il Game.

## Chess Problems

Un chess Problem è costituito da una `Board` iniziale, rappresentante la situazione in mediares che l'utente si trova a dover interpretare, e da una lista di mosse ordinate che rappresentano quelli che devono essere i Movement corretti per risolvere il problema.

I problemi sono affini alla modalità classica degli scacchi.

Per la loro creazione ho deciso di utilizzare il pattern **Factory**, questo mi ha permesso di generare molto semplicemente i vari problemi. Necessitando questi di una `Board` di partenza, esattamente come tutte le altre modalità di gioco, ho deciso di creare anche una `StartingBoardFactory`, utilizzata sia dalle altre modalità di gioco, sia dai vari problemi scacchistici per reperire le diverse `Board` di partenza per le varie modalità.

Per semplificare ulteriormente il compito alla Factory di `Board`, ho scelto di utilizzare il pattern **Builder** per la loro creazione.

Il manager dei movimenti per il problema scacchistico è stato descritto in precedenza. Proprio a quest' ultimo spetta il compito di verificare se la mossa effettuata dall' utente corrisponda ordinatamente a quella corretta secondo la lista definita dal problema.

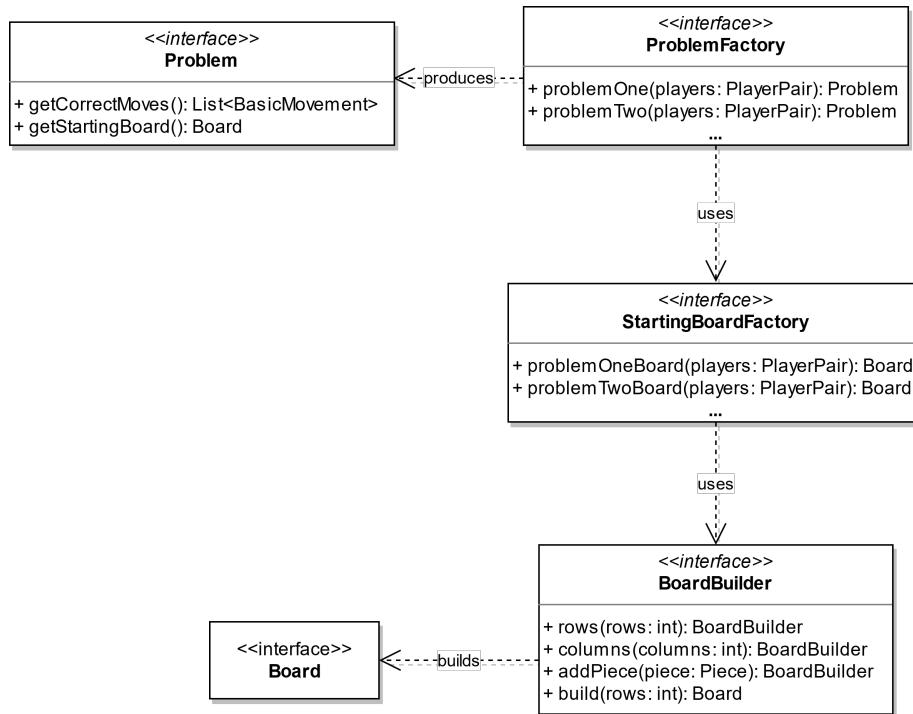


Figura 2.33: Gestione e creazione dei chess Problem

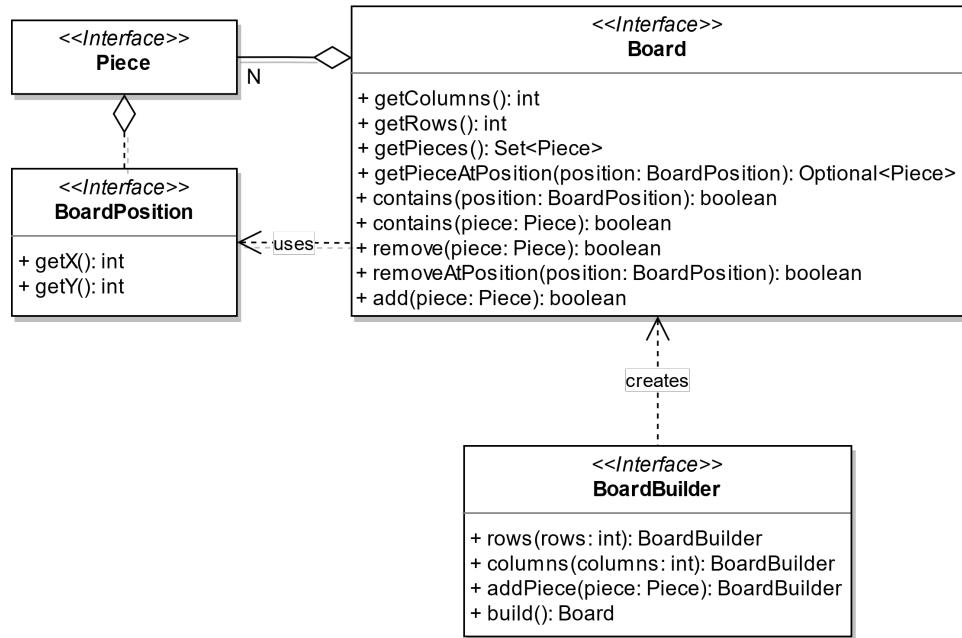


Figura 2.34: Focus gestione scacchiera

# Capitolo 3

## Sviluppo

### 3.1 Testing automatizzato

Sin dalle fasi iniziali abbiamo deciso di approcciarcici allo sviluppo seguendo una metodologia test-driven, utilizzando JUnit 5.

I test venivano scritti in modo tale cercare di "rompere" l'applicativo, così facendo siamo stati in grado di scovare problemi ed errori altrimenti difficilmente individuabili.

Come principio, nel caso un test non passasse, ci si soffermava ad individuare il problema per poi risolverlo. Non venivano aggiunte feature e non si proseguiva con lo sviluppo fino a che i test non passassero tutti.

Alla creazione di una feature, oppure di una sezione dell'applicativo, venivano creati i relativi test.

Soprattutto nelle fasi iniziali, dove c'era necessità di studiare gli edge-cases ed il funzionamento del motore, non avendo ancora implementato la GUI, i test sono risultati il modo più efficace e veloce per scovare inconsistenze altrimenti quasi impossibili da identificare. In più, quando implementata, è anche stato testato il funzionamento corretto della GUI.

#### 3.1.1 Andruccioli Manuel

Avendo realizzato qualche componente che va a scrivere su file, ho deciso fosse corretto testare il tutto. Infatti, nei test, viene salvata una copia dei dati già esistenti, in modo da poterla ripristinare alla fine degli stessi.

Avendo sviluppato lo scheletro della scacchiera grafica, ho pensato fosse corretto che venisse testata ed ho sfruttato la libreria `TestFX`.

Una prima parte è servita per testare le funzionalità basilari dell'applicazione.

Un test importante che ho realizzato è stato quello di `integration`: viene simulata un'intera partita di scacchi in modo random. Questo test ci ha permesso di scovare un bug che altrimenti non avremmo trovato facilmente: su `Windows` non dava problemi, mentre su `Linux`, dopo molte mosse, il gestore dei suoni dava errore e non faceva più crearene.

- UserTest

- UserBuilderTest
- UserStorageTest
- UserManagerTest
- ValidatorTest
- LeaderboardTest
- GameBoardTest
- **Integration:** GameBoardIntegrationTest

### 3.1.2 Scolari Stefano, Mazzoli Alessandro

Avendo sviluppato componenti dell'applicativo che per essere testate nella loro interezza necessitavano di un certo grado di mutualità, abbiamo deciso di collaborare a stretto contatto per la loro realizzazione.

Sarebbe sinceramente difficile dividere con rigore quali test siano stati sviluppati con assoluta indipendenza uno dall'altro e per questo abbiamo scelto di citarli come test comuni. Le componenti sono state testate in parallelo al loro sviluppo.

Il motore è stato strutturato in maniera **layered**, test di componenti successive andavano ad utilizzare componenti precedentemente già testate, garantendone la correttezza. Si nota infatti dai test, un progressivo grado di astrazione: partendo inizialmente dal poter testare semplicemente se un pezzo modificasse la propria posizione, fino al simulare situazioni di gioco, testandone il corretto **outcome**. Così facendo siamo arrivati ad analizzare anche il corretto funzionamento di alcune delle varianti implementate.

- MovementTest
- MovementManagerTest
- ClassicGameControllerTest
- PawnMovementVariantTest
- PieceSwappingVariantTest
- PieceTest
- BoardTest
- MatchTurnsTest
- ClassicGameTypeMatchTest

### 3.1.3 Patriti Tommaso

Testare la mia parte è stato fondamentale, infatti, dovendo lavorare con risorse del jar e file, le eccezioni e variabili di cui controllare integrità e correttezza sono state molteplici. Il test riguardante il **Timer**, è particolare, in quanto utilizza una sleep per verificarne il funzionamento.

- TimerTest
- SettingTest
- ReplayTest
- DynamicConfigTest

## 3.2 Metodologia di lavoro

Sviluppando il progetto si è cercato di mantenere più indipendenza possibile dal punto di vista del lavoro svolto, pur sempre confrontandoci quotidianamente per aggiornare l'uno l'altro dei progressi fatti sulle rispettive sezioni di lavoro.

La fase iniziale è stata di analisi, durante la quale abbiamo lavorato tutti e quattro a stretto contatto per definire attraverso grafici UML ed idee quella che doveva essere la nostra architettura.

Ci siamo soffermati per molte ore su questa fase, ed infatti l'implementazione concreta di quello che avevamo ideato e definito è risultato privo di particolari inghippi o problemi. Durante le fasi iniziali di sviluppo la collaborazione e confronto è stata molto intensa. Proseguendo con lo sviluppo ogni componente del gruppo ha lavorato con indipendenza alla propria parte.

### Workflow

Per organizzare bene il workflow di gruppo sul progetto abbiamo utilizzato Trello.com, dove nella bacheca da noi creata era possibile definire degli "step" e delle indicazioni su quali fossero le cose da gestire o implementare, il tutto seguendo un ordine gerarchico. Per ottenere ciò ci siamo basati sul **metodo KanBan** che prevede di avere 3 colonne:

- **To-Do:** l'insieme delle cose da fare non ancora iniziata.
- **Doing:** l'insieme delle cose che si stanno implementando.
- **Done:** l'insieme delle cose già completate.

In questo modo con Trello abbiamo gestito la creazione di Task che qualcuno avrebbe dovuto svolgere, infatti una volta creato un task lo si poteva assegnare ad uno o più persone. Avendo la colonna dei Doing si poteva sempre sapere chi stava lavorando a cosa evitando di finire a lavorare in parallelo alla stessa cosa.

La nostra estensione del KanBan prevedeva altre colonne quali **Bug Fix** e **General**, in BugFix venivano spostati tutti i Task completati ma che andavano controllati mentre su General c'erano gli appunti e le idee che venivano in mente ai membri del gruppo per migliorare il progetto.

### Version Control

Come DVCS è stato utilizzato Git.

Si è scelto di utilizzare una metodologia ispirata a GitFlow, dove il branch principale era 'main', dal quale sono state fatte le release.

Il branch di sviluppo di default è stato 'develop'.

Nel momento in cui vi era la necessità di sviluppare una nuova feature, si creava ogni volta un branch apposito denominato 'feature-\*', nel quale si aveva quindi la possibilità di implementare la funzionalità in un ambiente separato, senza rischiare di avere "collisioni" con il lavoro di altri membri del gruppo.

I vari componenti del gruppo hanno quindi lavorato sui branch delle feature in quel momento da loro sviluppate, per poi mergiare il tutto, una volta passati i test, sul branch 'develop'.

#### 3.2.1 Andruccioli Manuel

Realizzazione dell'intero sotto-sistema di salvataggio degli utenti con i loro rispettivi punteggi e possibilità di recuperarli, sotto forma di classifica. Funzionalità per validare la password degli utenti.

Creazione dell'utility per generare la cartella `.jhaturanga` nella home directory e controlli su di essa. Successivamente è stata ampliata da chi aveva necessità di aggiungere file al suo interno.

Realizzazione dello scheletro grafico della scacchiera con possibilità di drag-and-drop dei pezzi. Realizzazione della grafica relativa alla classifica. Caching di immagini e suoni.

#### 3.2.2 Mazzoli Alessandro

Realizzazione del motore scacchistico: gestione ed implementazione del concetto di **Game** e di **Match**.

Sviluppo dei componenti quali il **Player** ed implementazione della modalità online tramite MQTT.

Refactoring della grafica dell'applicazione e delle board della GUI tramite la struttura gerarchica delle **GraphicBoard**.

Gestione generale della struttura MVC.

### 3.2.3 Patriti Tommaso

Grafica volta al test dell'applicazione durante le prime fasi di sviluppo con fxml di base, timer e chiamate asincrone, salvataggio e replay delle partite, suoni della scacchiera, pezzi scacchiera personalizzabili, grafica applicazione personalizzabile, possibilità di poter installare dei resource pack esterni, salvataggio delle impostazioni preferite dall'utente e relativa gestione file json, binding dell'applicazione(resizable), struttura ramificata dei css e caricamento di questi ultimi, PageLoader per caricamento, switch, reload e passaggio della scena, gestione di alcune direcotry di .jhaturanga.

### 3.2.4 Scolari Stefano

Realizzazione del motore scacchistico: gestione ed implementazione `GameController`, implementazione movimenti e `MovementManager`, creazione strategie di movimento.

Implementazione ed ideazioni delle varianti e realizzazione dei problemi scacchistici.

Gestione `Piece` e `Board` iniziali.

Implementazione controller relativi a sezioni affini a quelle da me sviluppate ed implementazione grafica iniziale delle `Board` utilizzando JavaFx.

## 3.3 Note di sviluppo

### 3.3.1 Andruccioli Manuel

- **Programmazione Funzionale** durante lo sviluppo, soprattutto per la validazione delle password.
- **Optional** molto presenti in gran parte del mio codice, proprio per evitare null come valore di ritorno. Sono stati molto utili per aggiungere chiarezza al codice.
- **Generici**: sfruttati per definire un'interfaccia utile alla creazione di funzioni.
- **JSON**: libreria utilizzata per il salvataggio degli utenti su file Json.
- **Google Guava**: libreria utilizzata per l'Hashing delle password.
- **TestFX**: libreria utilizzata per il testing dell'interfaccia grafica.

### 3.3.2 Mazzoli Alessandro

- **Programmazione Funzionale** durante lo sviluppo di questo progetto ho potuto sfruttarla per quel che è inherente alla gestione del motore scacchistico. È risultato infatti di inimitabile aiuto il package `java.util.function` che ho avuto modo di approfondire durante il progredire del progetto.

- **Lambda** di paro passo con la programmazione funzionale è risultato necessario se non obbligatorio a livello stilistico utilizzare le lambda expression, le quali mi hanno permesso di avere del codice pulito, leggibile e chiaro.
- **Stream** ampiamente utilizzate di paro passo con quella che è stata la parte che fa uso della programmazione funzionale e delle lambda expression.  
Le ho trovate una delle feature più interessanti che Java offre e tuttora me le sto studiando sempre più nel profondo per capire come funzionano internamente.  
Per comprenderne ancora di più l'utilizzo ne ho inoltre creata un implementazione in Python creando una libreria.
- **Optional** data l'estrema utilità e flessibilità ne ho utilizzati molti nella mia parte di model. Ho notato che offrono una gran chiarezza nella gestione di possibili valori nulli e inoltre offrono metodi che permettono di sfruttare la programmazione funzionale quali map, orElse, ifPresent...
- **MQTT** avendo implementato la modalità online tramite protocollo MQTT è stato di fondamentale aiuto l'utilizzo della libreria paho-mqtt che mi ha permesso di implementare la comunicazione via internet molto velocemente.
- **Gradle** fin dall'inizio mi sono occupato del setup del progetto che fin da subito è stato deciso che sarebbe stato gestito tramite gradle. Durante il periodo di sviluppo ho avuto modo di approfondire, seppur poco, il suo funzionamento poichè lo trovo uno strumento molto potente e assolutamente indispensabile in un progetto dove si ha bisogno di librerie esterne. È risultato infatti di estrema facilità l'importazione di librerie tramite Gradle.

### 3.3.3 Patriti Tommaso

- **JavaFx** è stato il motore grafico della nostra applicazione, e ho tenuto molto a gestire la flessibilità grafica. Ogni pagina ha il suo personale file fxml e per mezzo dei binding l'applicazione si adatta a qualunque monitor desktop, mantenendo una continuità grafica in tutti i sistemi operativi.(In OSX e alcuni linux Desktop manager il full screen viene gestito in maniera particolare e ho lavorato per avere un PageLoader in grado di gestire ogni variante)
- **Gson**, libreria di google per la gestione di file json, è stata necessaria per la gestione dei file di configurazione dell'applicazione.
- **Reflections** è stata utilizzata per navigare agevolmente nelle risorse del file jar e poter duplicare dei file di default.
- **Optional** Nella mia parte di progetto ho lavorato con molti file, quindi per la gestione di eventuali file non presenti o per errori di formattazione, è stato obbligatorio l'utilizzo di Optional.

- **I tipi generici** sono stati utili per avere un' implementazione efficace di Strategy.
- **Runnable** sono stati utili per avere una comunicazione asincrona con l'applicazione senza side-effect generati dall'implementazione dei Thread da parte delle view.
- lambda e stream usate dove necessario

### 3.3.4 Scolari Stefano

- **Stream** Scrivere `Stream` ed utilizzarle nelle mie sezioni di progetto è stata una tra le parti che ho apprezzato maggiormente, a tal punto che ho deciso in alcune specifiche parti di utilizzare una libreria esterna chiamata "StreamEx", che nasce come "enhancement" delle `Stream` classiche fornite da Java.  
Avendo lavorato principalmente alla sezione del motore scacchistico, interfacciarmi con `Collection` oppure necessitarne la creazione era una costante.  
Le `Stream` sono quindi risultate utilissime, ed infatti sono presenti in quasi tutte le classi da me scritte.
- **Generici** Perché il package `java.util.function` non fornisce una `TriFunction`, ho avuto necessità di scrivere una semplice `Functional Interface` che utilizza quindi i generici, lo stesso vale per il `TriPredicate`.
- **Lambda expressions** Ho fatto largo uso delle lambda expressions, sia quando trattavo le `Stream`, sia quando dovevo definire ed utilizzare interfacce funzionali. Ho anche cercato di utilizzare il `Method Reference` ove possibile.
- **Programmazione funzionale** Pur non avendo affrontato questo argomento a lezione, durante le fasi di sviluppo ho avuto modo di imparare strada facendo ed utilizzare ampiamente il package `java.util.function` e le rispettive interfacce funzionali. Ho in molti punti definito `Predicate`, `BiPredicate`, `Supplier`, `Consumer`, `Function` e `UnaryOperator`, infatti le strategie di movimento dei pezzi, alla base, ne fanno largo uso.
- **Optional** Ho utilizzato molto spesso `Optional` ed i metodi di questa classe (`map`, `orElseGet`, etc) quando vi era la necessità di lavorare con valori potenzialmente nulli.

### 3.3.5 Crediti

- **Page** L'idea dell'entità `Page` è stata ispirata dalla classe `GameScene` del progetto OOPang
- **PageLoader** L'idea del componente `PageLoader` è ispirata dalla classe (`SceneLoader` del progetto OOPang.

# Capitolo 4

## Commenti finali

### 4.1 Autovalutazione e lavori futuri

#### 4.1.1 Andruccioli Manuel

Sono molto soddisfatto del lavoro svolto, sia da me, che dal mio team. Fin dalle fasi di analisi, abbiamo avuto grande capacità comunicativa, esponendo i nostri dubbi, pensieri e motivazioni. Questa armonia nel gruppo ha generato un flusso di lavoro ottimale per procedere nella realizzazione del progetto.

Seppur non abbia partecipato alla logica del motore scacchistico, proprio perché non sarebbe stato possibile dividerci quella parte in 4, mi sono ritagliato una modesta sezione, dove ho gestito, sempre in ottica di estensibilità, la possibilità di salvare gli utenti e i suoi relativi dati.

L'utilizzo corretto di **Git**, ci ha permesso un lavoro agile, pensando ognuno alla sua parte, per poi unirle insieme una volta terminato lo sviluppo. Questo ciclo di vita ci ha permesso di incrementare gradualmente le funzionalità.

Questa metodologia di lavoro mi ha permesso di comprendere le difficoltà dei progetti di dimensione maggiore a ciò che si può/riesce a vedere in aula.

Ringrazio il mio team per il lavoro svolto.

#### 4.1.2 Mazzoli Alessandro

Fin dall'inizio mi sono occupato della gestione architetturale del progetto a livello della struttura dei package, costruita in modo da avere una gerarchia in modo che sia molto semplice trovare una interfaccia/classe quando se ne ha bisogno.

Questo è stato fatto fin dall'inizio in vista di un possibile incremento molto elevato del numero di classi, come infatti è successo.

Ci si è ritrovati infatti alla fine con un elevato numero di classi che tuttavia però era agevolmente gestibile poiché avevamo una solida gerarchia dei package.

Essendo io un appassionato di Clean Code un altro aspetto molto importante in cui ho aiutato è stata consigliare i miei compagni sulle "regole" da seguire in modo da avere un progetto e un codice pulito e uniforme.

Durante tutta la parte di sviluppo dell'applicazione ho utilizzato ampiamente la tecnica del refactoring, riguardando il codice da me scritto e le scelte da me fatte analizzandole criticamente e cercando di capire in che modo si potesse migliorare l'architettura da me creata.

Questo ha portato ad un continuo refactor che mi ha permesso di rendere la mia parte di codice a mio parere molto esplicativa e ben strutturata.

Molto di aiuto in questa fase è stato l'utilizzo del plugin CheckStyle che ho trovato di estrema utilità nella creazione di software pulito ed uniforme.

Parte fondamentale è stato il continuo utilizzo di UML che mi ha aiutato ad avere una visione d'insieme del contesto sui cui stavo lavorando in modo da poter ad esempio scomporre un'entità più grossa in un insieme di più entità di dimensione inferiore.

Altro tool fondamentale è stato l'utilizzo di Git, che ho avuto di modo di imparare ad usare abbastanza bene. Tuttavia spero in futuro di poterlo studiare più a fondo potendone utilizzare le feature più avanzate che a non ci sono state insegnate a lezione.

Durante lo sviluppo del progetto Jhaturanga ho avuto modo di venire sempre di più a contatto con quello che è il mondo della programmazione funzionale da cui sono rimasto affascinato.

Spero di avere modo di approfondirla ulteriormente studiandola insieme alla Reactive Programming poiché le trovo altamente entusiasmanti e interessanti.

#### 4.1.3 Patriti Tommaso

Questa attività è stata importante per la mia crescita, soprattutto dal punto di vista progettuale. Purtroppo sono entrato nel vivo dell'analisi del progetto in un secondo momento rispetto ai miei compagni, e questo nella fasi intermedie di sviluppo ha determinato scarsa produttività nelle parti più interne(model). Nelle fasi finali sono riuscito a dedicarmi al progetto in maniera più concreta, una volta fatta un'ulteriore analisi delle mie parti, sono andato a sistemare e correggere il codice scritto in precedenza e a gestire in maniera più efficiente le ultime funzionalità da me aggiunte, integrandole bene con il lavoro svolto dai miei colleghi.

Sinceramente non sono pienamente soddisfatto del mio operato, un mio maggior coinvolgimento nelle parti iniziali di progettazione e studio dei Pattern avrebbe sicuramente portato il mio lavoro ad un gradino più alto, anche se credo di aver avuto un buon incremento qualitativo nelle ultime fasi del progetto.

Un aspetto molto positivo è stato un buon utilizzo di Git, che usavo già da diverso tempo ma in maniera inappropriate.

Per quanto riguarda il lavoro d'équipe, la comunicazione all'interno del gruppo è stata serena e lineare, anche per merito di numerose sessioni d'esame affrontate già precedentemente assieme che ci hanno permesso di trovare un ottimo grado di affiatamento e collaborazione.

Per il futuro conto di continuare a studiare in maniera approfondita la programmazione ad oggetti, integrandola possibilmente con nuovi linguaggi.

#### **4.1.4 Scolari Stefano**

Lavorare a questo progetto mi ha permesso di affinare maggiormente le mie conoscenze di Java e di approccio allo sviluppo.

Prima di sviluppare questo applicativo mi era totalmente estraneo come gestire un progetto di tali dimensioni lavorando in team.

Mi ritengo molto soddisfatto del lavoro svolto e delle nuove competenze acquisite. Sono molto felice della comunicazione e della generale gestione del lavoro da parte del gruppo. Ritengo di essermi impegnato molto per la buona riuscita di questo progetto, ho cercato di adottare le metodologie di lavoro migliori possibili ed ho sempre tentato di scrivere codice al massimo delle mie capacità.

Come citato nella sezione precedente, durante lo sviluppo ho avuto modo di approcciarmi, se pur a livello basico, ad alcuni concetti ed aspetti della programmazione funzionale che mi sono risultati utili per scrivere più agilmente il codice.

Il mio obiettivo per il futuro è approfondire diversi aspetti della programmazione ad oggetti, studiando in maniera ancora più profonda quali siano le good-practice di Java, le tecniche di sviluppo e l'uso dei Pattern.

Durante le fasi di sviluppo mi sono appassionato a queste tematiche, leggendo molto a riguardo e guardando diversi talk sull'argomento(e.g.: Devoxx Talk).

Sono inoltre molto curioso di conoscere meglio il mondo della programmazione funzionale.

## **4.2 Difficoltà incontrate e commenti per i docenti**

### **4.2.1 Andruccioli Manuel**

Ritengo indubbiamente che questo sia uno dei migliori corsi mai frequentati finora. La sua completezza, profondità, precisione, mi ha portato ad acquisire qualità non da poco.

Il fatto che non sia riuscito a superare l'esame al primo tentativo, mi ha portato a passare due settimane intere sulla programmazione funzionale, il che è stato davvero importante per acquisire, in modo non superficiale, competenze utili sia in Java, ma anche in altri ambiti. Tornando su progetti personali (e.g. che utilizzano Javascript), ho potuto subito apprezzare l'utilizzo della programmazione funzionale.

Ringrazio i professori per la qualità e l'impegno speso per la realizzazione di questo corso.

### **4.2.2 Mazzoli Alessandro**

Sono stato molto soddisfatto dalle conoscenze appreso dal corso di Programmazione Ad Oggetti, lo reputo assolutamente il corso più valido visto finora.

Di grande importanza è stata la capacità di spiegazione dei docenti sia di teoria che di

laboratorio che si sono rivelati essere di un eccellente preparazione riuscendo a rispondere a qualsiasi mio dubbi anche se questo andava molto nello specifico.

L'apprendimento è risultato quindi semplice e gradevole dato l'elevata qualità delle spiegazioni fornite che miravano ad estirpare qualsiasi dubbio degli studenti.

Inoltre una nota di merito va al fatto che vengano insegnati i Design Pattern, e si cerca inoltre di far passare agli studenti un po di enfasi verso l'arte del clean code.

Le uniche pecche che ho potuto trovare sono state forse la spiegazione di JavaFx e di Gradle.

Entrambi questi argomenti sono stati spiegati a lezione, tuttavia JavaFX è stata trattato molto velocemente senza particolari approfondimenti, quindi per sviluppare il progetto è stato necessario studiarne il funzionamento in autonomia.

Tuttavia il cercare l'indipendenza degli studenti nello studiarsi un framework la trovo un ottima pratica per prepararli a quello che sarà poi il mondo del lavoro.

Relativamente a Gradle, che ho trovato un po' più complesso da studiarsi rispetto a JavaFx, l'unica nota che vorrei segnalare è che la spiegazione di questo tool viene fatta verso metà anno, quando purtroppo gli studenti non sono ancora nell'ottica del progetto.

Reputo quindi che sarebbe stato meglio farne una spiegazione leggermente più approfondita verso fine del corso, enfatizzandone l'importanza nel progetto e mostrando qualche esempio di utilizzo più avanzato.

#### **4.2.3 Patriti Tommaso**

A mio avviso questo è stato il corso più serio affrontato fino ad ora. Per me è stato molto importante in quanto è andato a spaziare vari aspetti della programmazione e mi ha fatto vedere quest'ultima con una luce differente, nel bene e nel male. Questo corso, i laboratori e soprattutto il progetto svolto con i miei compagni sono stati dei tasselli importanti che sicuramente mi hanno aiutato a capire quale strada voler prendere nella giungla dell'informatica.

Probabilmente il maggior scoglio da me affrontato è stato quello di applicare le buone regole del "clean code" e perdere le cattive abitudini.

#### **4.2.4 Scolari Stefano**

Quello di Programmazione ad Oggetti è stato il corso che ho seguito con maggior interesse e che maggiormente mi ha appassionato.

Le lezioni tenute durante il corso sono sempre state chiare e precise. L'attenzione, professionalità e nitidezza con cui il professor Viroli ha tenuto il corso, mi ha in qualche modo spinto a cercare di dare il massimo sia durante l'esame sia lavorando al progetto.

Questo corso è l'unico fino ad ora che mi ha portato a voler approfondire e studiare indipendentemente temi legati alla materia, andando oltre quello che mi era stato spiegato a lezione.

L'utilizzo di Pattern, il dare importanza alla "qualità" del software sviluppato, le tecniche avanzate del linguaggio, i principi di buona programmazione, le metodologie di approccio al lavoro e di sviluppo sono tutti temi che voglio approfondire, sia individualmente che accademicamente.

# Appendice A

## Guida utente

### A.1 Selezione Piattaforma

All'avvio dell'applicazione, lanciando il comando 'java -jar nomeJar.jar', verrà chiesto all'utente di scegliere attraverso quale piattaforma grafica interagire con il software, terminale oppure GUI.

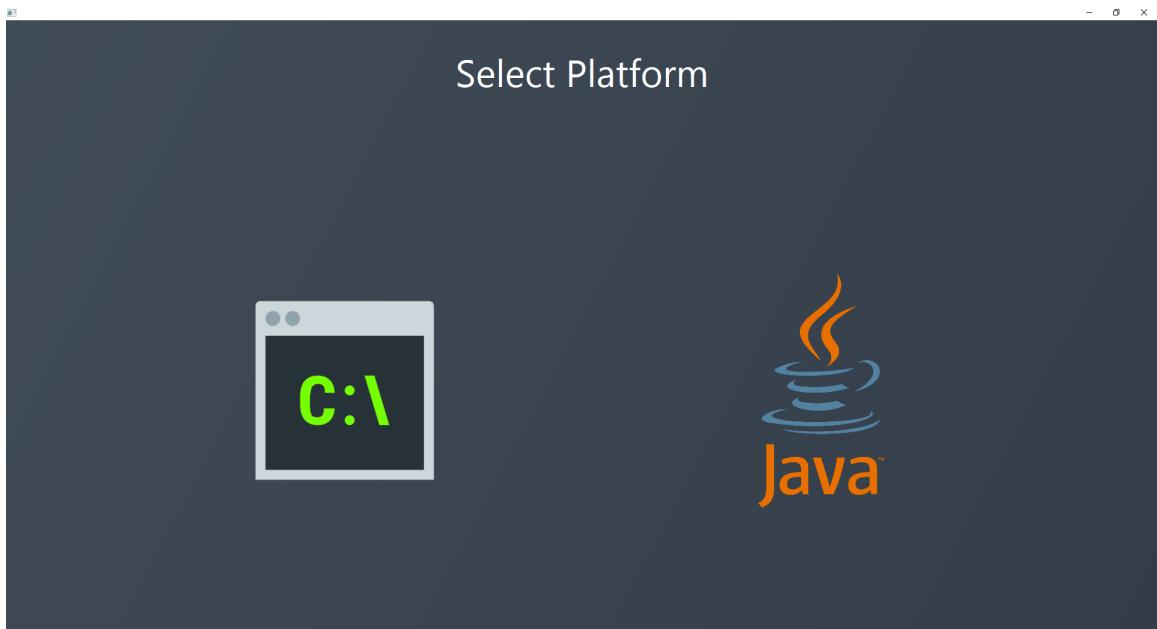


Figura A.1: Schermata iniziale per scelta piattaforma grafica

Poiché la versione da riga di comando è un semplice proof-of-concept, come precedentemente detto, non rappresenta l'applicativo nella sua interezza, ed è quindi meno fornita di opzioni e feature rispetto alla controparte da GUI.

Anche per questo motivo non ci si è occupati di rendere portabile la versione da terminale, sfortunatamente infatti non è possibile visualizzare né da CMD né da Powershell i caratteri rappresentanti i simboli degli scacchi.

Se si vuole comunque testare l'applicativo nella sua versione command-line da Windows sarà necessario settare la console in modo tale da supportare i colori ed un font che contenga i simboli dei pezzi degli scacchi.

Per settare correttamente la console:

- Si segua questa soluzione per abilitare la visualizzazione dei colori da Powershell e CMD <https://stackoverflow.com/a/51681675/13800588>
- Si segua poi invece questa guida per aggiungere un nuovo font(si consiglia DejaVu Sans) <https://maketecheasier.com/add-custom-fonts-command-prompt-windows10/>

Ci concentreremo ora a guidare l'utente nell'uso della versione GUI, in quanto quella da command-line risulta essere intuitiva.

## A.2 Login

Una volta scelta la modalità di gioco da GUI verrà richiesto di eseguire il login, oppure la registrazione. Nel caso non si sia interessati a registrarsi e tenere traccia dei punti si può saltare questo passaggio premendo il pulsante 'Log as Guest'.

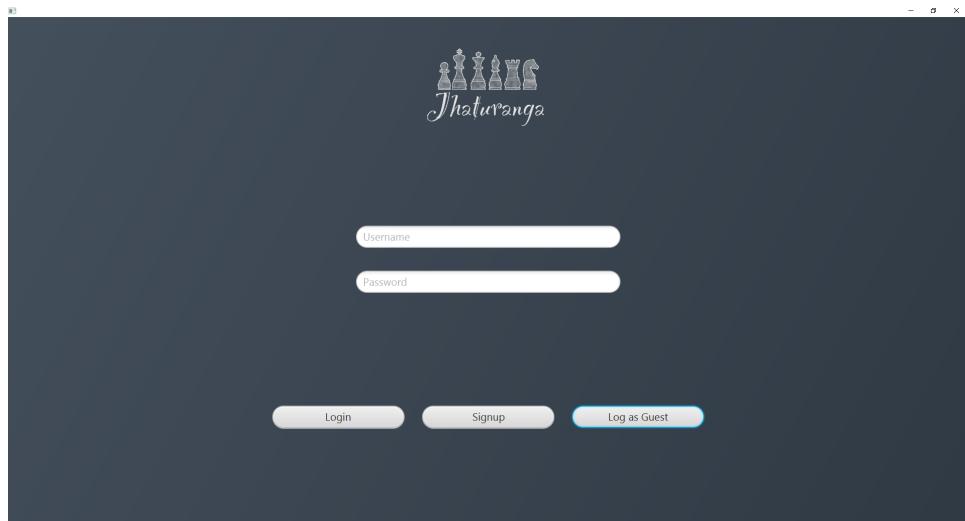


Figura A.2: Schermata di login

## A.3 Home Page

Superata la schermata di login ci si troverà nella sezione principale dell'applicativo, da cui è possibile iniziare una nuova partita, visualizzare la leaderboard, visualizzare i replay delle partite giocate oppure accedere alle impostazioni.

Inoltre in alto a destra saranno presenti gli username dei due utenti loggati, di default il secondo user è un guest. Per cambiare utente basterà cliccare sulla label corrispettiva e si aprirà la pagina di login.

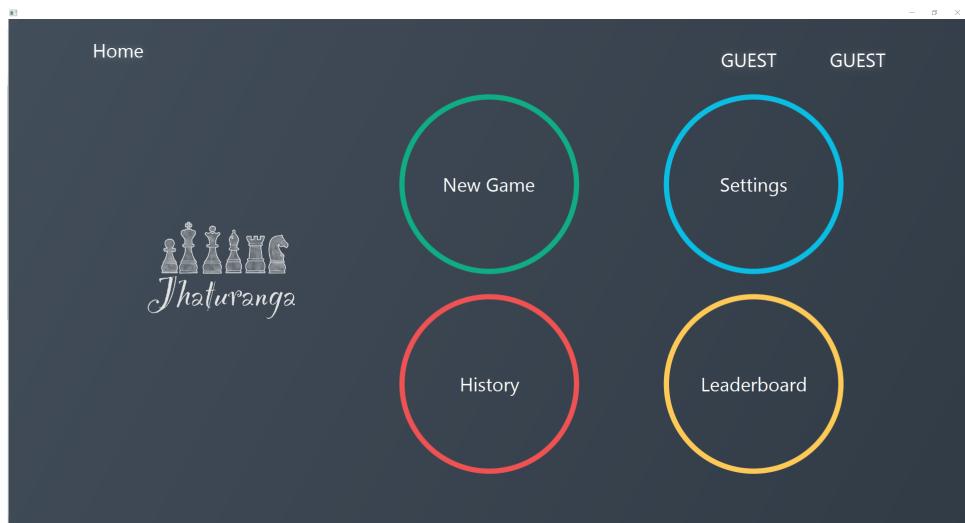


Figura A.3: Schermata selezione modalità

## A.4 Settings

Dalla pagina delle impostazioni sarà possibile cambiare i vari stili dell'applicazione tra i quali:

- Tema dell'applicazione
- Stile dei pezzi
- Volume dell'applicazione

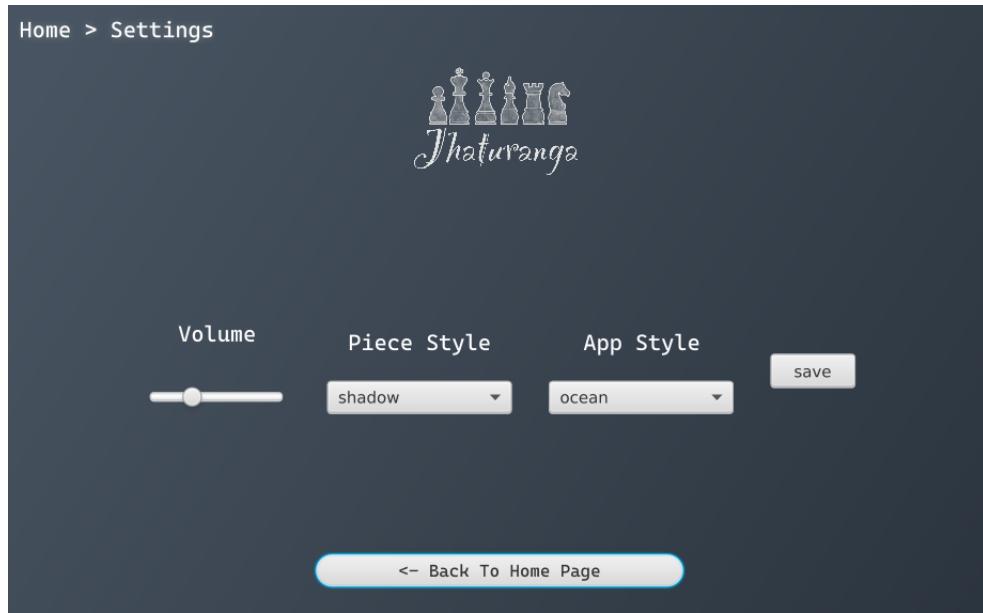
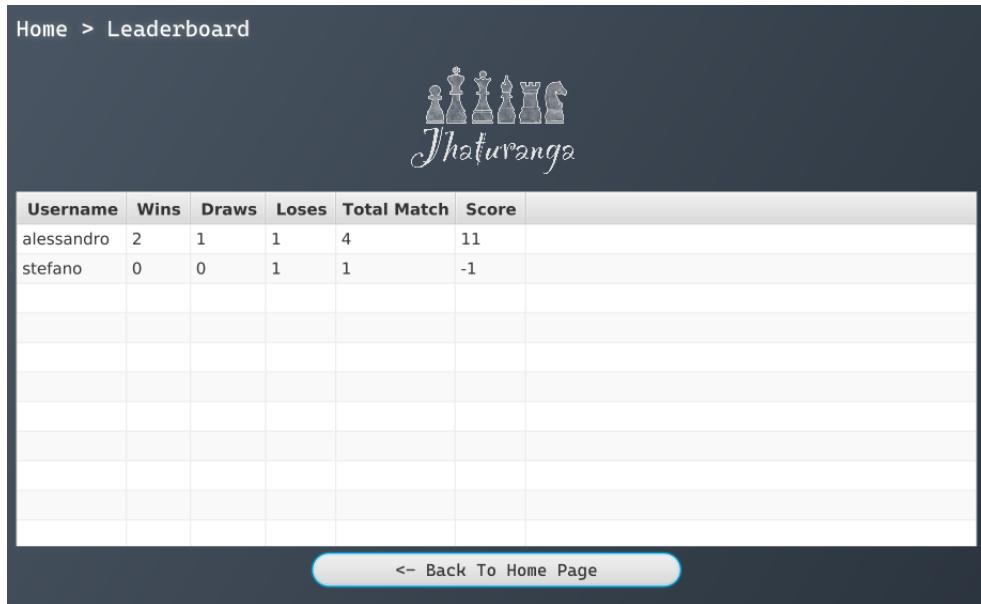


Figura A.4: Schermata selezione modalità

## A.5 Leaderboard

Dalla pagina della leaderboard sarà possibile visualizzare la classifica degli utenti salvati. Cliccando sulle varie header tab tipo "Wins" e "Loses" si potrà ordinare gli utenti in ordine crescente/descrescente relativamente a quel carattere.



The screenshot shows a dark-themed user interface for a chess variant named "Jhaturanga". At the top, there's a navigation bar with "Home > Leaderboard". Below it is a decorative header featuring a row of chess pieces and the word "Jhaturanga". A table displays the current leaderboards:

Username	Wins	Draws	Loses	Total Match	Score
alessandro	2	1	1	4	11
stefano	0	0	1	1	-1

At the bottom, a button labeled "<- Back To Home Page" is visible.

Figura A.5: Schermata selezione modalità

## A.6 History

Dalla pagina History sarà possibile visualizzare l'elenco delle partite salvate e sceglierne una di cui visualizzare il replay. Una volta cliccato il bottone per visualizzarlo si aprirà quindi la pagina in cui verrà avviata la partita. Per navigare all'interno dello storico delle mosse si utilizzino i tasti analoghi a quelli utilizzati in-game che verranno spiegati nel capitolo dell'interfaccia di gioco.

Home > History



Game Type ▲	White Player	Black Player	Date	Action
1-Dimension	alessandro	GUEST	2021/04/17 - 09:30	<button>View Replay</button>
3-Col	alessandro	GUEST	2021/04/17 - 09:27	<button>View Replay</button>
Bombastic	alessandro	GUEST	2021/04/17 - 09:28	<button>View Replay</button>
Classic	GUEST	GUEST	2021/04/16 - 09:04	<button>View Replay</button>
Classic	GUEST	GUEST	2021/04/16 - 09:09	<button>View Replay</button>
Classic	GUEST	GUEST	2021/04/16 - 09:12	<button>View Replay</button>
Classic	GUEST	GUEST	2021/04/16 - 09:13	<button>View Replay</button>
Classic	GUEST	GUEST	2021/04/16 - 09:13	<button>View Replay</button>

<- Back To Home Page

Figura A.6: Schermata selezione modalità

## A.7 Nuova Partita

Nel caso si voglia iniziare una nuova partita verrà chiesto all'utente se la si vuole giocare in locale oppure online.



Figura A.7: Selezione offline oppure online

## A.8 Modalità Online

Nel caso la si giochi online si potrà scegliere se creare una partita oppure se unirsi ad una partita attraverso un codice. Se si sceglie di crearne una si viene inviati alla pagina di selezione della modalità di gioco e una volta cliccato il bottone **Play** dopo qualche secondo comparirà un popup con il codice del Match che andrà fatto inserire al giocatore che si vuole unire alla partita.

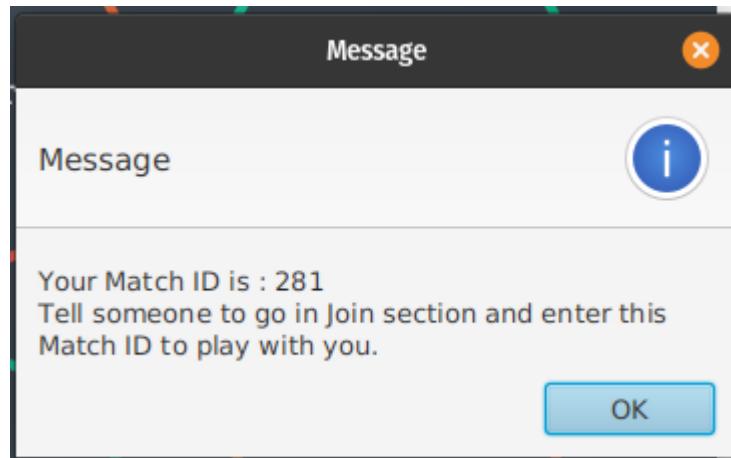


Figura A.8: Schermata selezione modalità

## A.9 Modalità Offline

Se invece si vuole giocare offline saranno presenti tre opzioni: la select game type, la customized board e la chess problems.

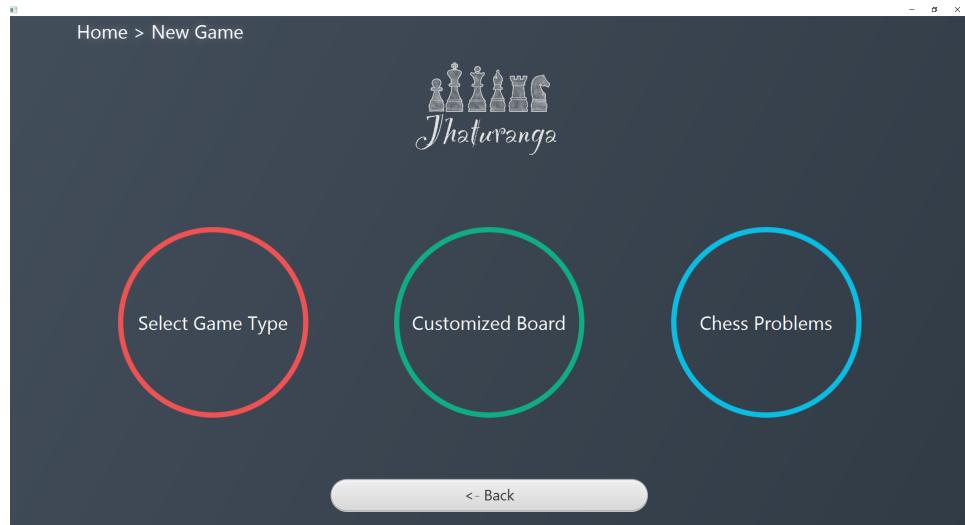


Figura A.9: Menù selezione game type

## A.10 Selezione Modalità

Dalla prima è possibile selezionare una delle molte varianti scacchistiche. Si usa la rotella del mouse per navigare fra le varie modalità.

Nella tab di destra è possibile selezionare il timer da utilizzare per la partita ed a chi assegnare il colore bianco.

Una volta premuto su play inizierà la partita.

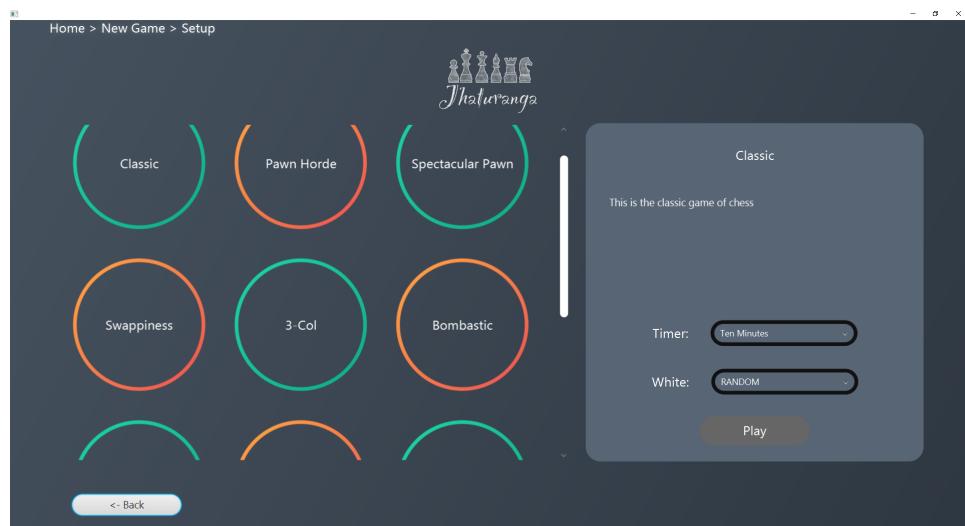


Figura A.10: Selezione del game type

## A.11 Interfaccia Di Gioco

L'interfaccia di gioco si presenta nel seguente modo, il giocatore bianco si trova nella posizione bassa della scacchiera ed ha il rispettivo username + timer in basso a sinistra, il giocatore nero invece si troverà nella parte alta della scacchiera con il rispettivo username + timer in alto a destra.

Cliccando su un pezzo ci viene mostrato dove quel pezzo si può muovere marchiando le posizioni valide.

Per muovere i pezzi serve trascinarli e rilasciarli nella posizione di destinazione.

Una volta effettuata una mossa questa rimarrà evidenziata tramite la colorazione della cella di partenza e quella di arrivo fino a che il giocatore avversario non farà la sua mossa.

Mentre si gioca è possibile visualizzare lo storico delle mosse, muovendosi avanti(tasto "D" oppure "->") ed indietro(tasto "A", oppure "<-").

Con il tasto "resign" è possibile arrendersi e terminare la partita.

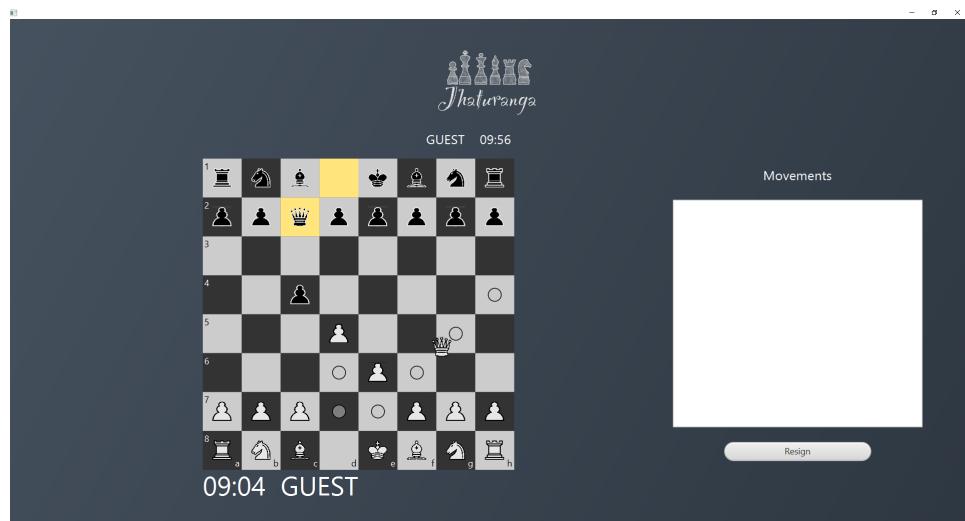


Figura A.11: Schermata di gioco

## A.12 Scacchiera Personalizzata

Dalla seconda invece si apre una board editor per creare la propria scacchiera di partenza customizzata. Le dimensioni della scacchiera sono ridimensionabili tramite i due prompt per inserire il numero di righe e colonne ed il tasto "resize" per applicare le modifiche. I pezzi vanno trascinati dalla sezione di destra e rilasciati sulla board tramite il LMB. Per rendere più veloce l'aggiunta dello stesso pezzo è possibile tenere premuto CTRL mentre si trascina un pezzo per "disegnare" la tastiera di quel pezzo.

Per rimuovere pezzi si utilizzi il tasto destro, sia cliccando individualmente sulla casella interessata, sia tenendo premuto mentre si trascina.

Una volta che si decide di aver ultimato la creazione della propria board si prema il tasto "Start" per iniziare la partita che utilizzerà la scacchiera appena creata come board di partenza.

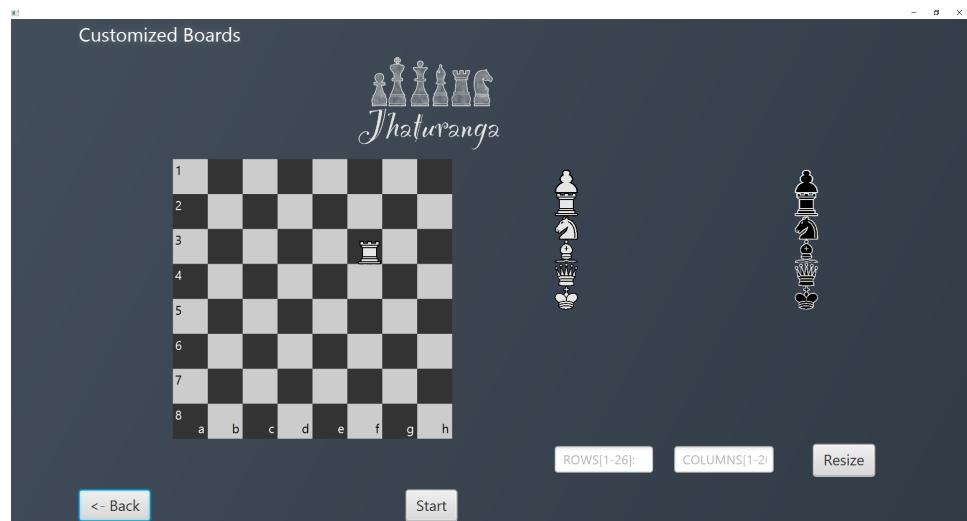


Figura A.12: Schermata per l'editing della board

## A.13 Problemi Scacchistici

La sezione per i problemi scacchistici risulta analoga a quella per la selezione dei game type.

## A.14 Resource Pack

Per aggiungere uno stile all'applicazione è possibile aggiungere figlio di stile (css) nella cartella .jhaturanga situata nella home directory. Più precisamente il file andrà aggiunto nella sottodirectory res/css/themes. Per aggiungere invece uno stile concernente la grafica

dei pezzi, bisogna aggiungere una cartella contenente le immagini di questi ultimi nella sottodirectory res/pieces. Andrà seguito il pattern degli stili già presenti seguendone quindi la denominatura e il formato dei file.

# Appendice B

## Esercitazioni di laboratorio

### B.1 Andruccioli Manuel

- Laboratorio 04: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p101058>
- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101061>
- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p100885>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p100887>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p102770>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p103801>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106682>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=66463#p106435>

### B.2 Mazzoli Alessandro

- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101104>

- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p100892>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p100894>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p102831>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p104182>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106622>
- Laboratorio 11: <https://virtuale.unibo.it/mod/forum/discuss.php?d=66463#p106266>

### B.3 Patriti Tommaso

- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p101540>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p101567>
- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p103917>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p104080>

### B.4 Scolari Stefano

- Laboratorio 04: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p101107>
- Laboratorio 05: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101105>
- Laboratorio 06: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p101037>
- Laboratorio 07: <https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p100910>

- Laboratorio 08: <https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p102771>
- Laboratorio 09: <https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p103968>
- Laboratorio 10: <https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106627>